Pricing multi-windowed barrier options using finite element method

Chengshi Ai

May 2013

A thesis submitted for the degree of Master of Philosophy
of the Australian National University

Australian
National
University

Declaration

The work in this thesis is my own except where otherwise stated.

Chengshi Ai

Acknowledgements

I would like to thank my supervisor Dale Roberts for all the help and support he has given. His extensive experience and inspiration is of great help in completing the thesis. I also appreciate that my friends Evelyn and Arthur provided valuable opinions on the draft of this thesis as well as the mental support during this process. I give thanks to my parents for their understanding and patience. Last but not least, I thank God for preserving my soul and guiding my step. The strength that He gives enables me to endure my frustrations, overcome difficulties, and then achieve.

Abstract

In this thesis we study pricing multi-windowed barrier options under three different models: Black-Scholes' model, Heston model, and the multi-dimensional Heston model proposed by De Col, Gnoatto and Grasselli. The PDE approach is employed where the option price is deemed as the solution of a partial differential equation.

The PDEs arising in the area of option pricing are most parabolic equations. The interesting questions are a) how to deal with the semi-infinite boundary; b) how to determine the boundary conditions when the domain changes with time. Especially we also consider the situation where the Feller condition is violated in the foreign exchange markets, which gives degenerate parabolic equations. We use the finite element method to obtain the numerical results of the PDEs. It is implemented by C++.

The main result of this thesis provides a practical scheme in pricing options in a real market scenario. All the coefficients used in the multi-dimensional Heston model can be calibrated once for all according to the real markets. Then the model dimension can be reduced when different types of options are priced.

Contents

Notation and terminology

Notation

$\mathbf{1}_{\{X\}}$	indicator function
$C_c^\infty(D)$	set of infinitely often continuously differentiable function on D with compact support
$E_{\mathbb{P}}$	expectation under the measure $\mathbb{P}$
$F_{t;T}$	forward price to time T at time t
$H^k(D)$	$:= W^{k,2}(D)$
$H_0^k(D)$	set of functions from $H^k(D)$ with vanishing trace on ∂D
$L^2(D)$	set of Lebesgue-measurable functions whose second power of their absolute value is Lebesgue-integrable on D
$\mathbb{P}$	a probability measure
$\mathbb{Q}$	a probability measure
S_t	asset price process
W_t	Brownian motion
$W^{k,2}(D)$	set of k-times weakly differentiable functions from $L^2(D)$, with derivatives in $L^2(D)$
Z_t	Brownian motion
∇	$:= \left(\frac{\partial}{\partial x_1}, \ldots, \frac{\partial}{\partial x_d}\right)^T$ Nabla operator
Φ	normal distribution function

$\|\cdot\|_V$	norm in the function space V
$\|\cdot\|_V$	semi-norm in the function space V
$(\cdot,\cdot)$	scalar product in L^2
∂D	boundary of the set D
$\overline{D}$	closure of the set D
$\vec{\mathbf{n}}$	outer unit normal vector with respect to the set D

Chapter 1

Introduction

Barrier options are a class of path-dependent options actively traded in the financial markets around the world. Unlike the vanilla options, barrier options depend not only on the final price of the underlying asset but also on the event that the price has crossed some barrier level before the expiration date. There are two basic types: "knock-ins" and "knock-outs", each having two subtypes,

1 *knock-ins*

   (a) the barrier is *up-and-in* if the option is only active if the barrier is hit from below.

   (b) the barrier is *down-and-in* if the option is only active if the barrier is hit from above.

2 *knock-outs*

   (a) the barrier is *up-and-out* if the option is worthless if the barrier is hit from below.

   (b) the barrier is *down-and-out* if the option is worthless if the barrier is hit from above.

In this thesis we will be concerned with a slight variant of the barrier option that is sold over-the-counter (OTC) by many banks: a multi-windowed barrier option. The barriers of such options can change discretely at anytime during the life of the option and even for some time windows can be deactivated. We consider multi-windowed barrier options in the foreign exchange (FX) market as this type of option is often used there.

1.1 Literature review

The first pricing formula of a continuously monitored European down-and-out option has been given by Merton [35] in 1973. Then in 1991 and 1994, both Rubinstein and Reiner [46] and Rich [43] presented the closed-form formulae of other types of standard European barrier options (i.e., call or put options which are either knock-in or knock-out).

Carr [8] also noticed that "Standard barrier options are now so ubiquitous that it is difficult to think them as exotic". This motivated him to value two extensions of the European down-and-out call options. The first of them is a barrier option with an initial protection period during which the option cannot be knocked out. Technically it can be considered as a particular case of a multi-windowed option; the main topic of this thesis. The second one is that the option is only knocked out when a second asset touches the barrier, which is also called "contingent option" or "rainbow barrier option".

Barrier options with initial protection have also been investigated by Heynen and Kat [24] and moreover they considered the situation where the barrier would change during the life of the option. Rogers and Zane [45] examined knock-out options with smoothly moving barriers and they sought a way to reduced the problem to one with a fixed barrier.

The pricing formulae of double barrier options where the barrier could be curved have been given by Kunitomo and Ikeda [33] through generalizing the Bachelier-Levy formula which gives the density of the first-passage time of Brownian motion over the boundary. Geman and Yor [19] obtained the Laplace transformation of the value of the double barrier options with two fixed barriers. Armstrong [3] extent Heynen and Kat's [24] formula and derived a formula which can be applied on the single-windowed barrier options.

There also have been many researchers working on numerical methods for barrier options. Lattice methods (i.e., binomial/trinomial trees) are possibly the most well-known and widely used methods for pricing vanilla options. Boyle and Lau [6] considered the binomial tree method for barrier option pricing and found that the convergence can be very poor if the barrier cannot be ensured to lie on a horizontal layer of the nodes in the tree. Also they proved that the method could cause persistent errors in barrier option pricing.

Ritchken [44] stated that the trinomial tree method has certain advantages over the binomial tree method in a sense that it provides more flexibility to ensure the nodes line up with the barriers. Both Boyle and Lau [6] and Ritchken [44]

concluded that the naive lattice methods can not be applied directly to complex barrier option pricing. Even though they employed modified algorithms, they cannot deal with the case where the barrier is too close to the initial asset price.

Cheuk and Vorst [9] proposed a modification of Ritchken's algorithm to alleviate the problems arising from the barrier being too close to the initial asset price. They applied their approach to many complex barrier option pricing cases, such as rainbow barrier options and simple moving barrier options. Even so, the number of time steps required by this algorithm can still be very large.

Lattice methods are actually thought to be in the framework of explicit finite difference (FD) schemes even though in the literature above they treated them as distinct techniques. Boyle and Tian [7] presented a modified explicit finite difference method to price barrier options. They solved the issue of aligning the nodes with barriers by constructing a grid lying on the barriers. Still, their method is laborious to solve discretely changing barrier options.

Figlewski and Gao [17] proposed an adaptive mesh technique to value barrier options, which is another method belonging to the class of trinominal tree methods. The basic idea is to use a fine mesh in regions where it is required and then transplant the computed results to the regions with a coarse mesh. While it provides much more flexibility and efficiency than the methods before, it has to satisfy the restriction that the points on the fine and coarse grids have to line up.

Monte Carlo methods are another popular tools in option pricing. Glasserman and Staum [21] illustrated how the standard Monte Carlo simulation causes bias and is not efficient when used for barrier option pricing. To resolve this problem, they introduced a modified technique that consists of changing measure at each step of the simulation and estimating the conditional probability when the conditional distributions are unavailable at some steps. Metwally and Atiya [36] presented another modification of the Monte Carlo method to achieve a fast and unbiased simulation. They supposed that the underlying security follows a jump-diffusion process and supplied an algorithm for deriving the probability density of the barrier-crossing time to determine if the crossing happened between two generating paths.

Although the Monte Carlo simulation is ideal for high-dimensional cases due to the fact that its complexity grows linearly with the number of dimensions, we can see that it needs to be adjusted to satisfy the requirements of different options. Moreover the Monte Carlo simulation is not efficient to compute the hedge parameters (i.e., Greeks) which are vital tools in financial risk management.

Results outlined above generally suffer one or more drawbacks. First of all, most suppose that the underlying asset price dynamics follow the classic Black-Scholes' framework [5] where the volatility is assumed to be constant. In the financial markets it has been empirically observed that with the time and the strike price of the underlying asset changing, the implied volatility has different values. The phenomenon is termed "volatility smile". For this reason, Dupire [13] proposed the local volatility model and Heston [23] proposed the stochastic volatility model.

Secondly, the barriers considered are often supposed to be constant or the continuous functions of time. There is not any financial reason to apply such restricted conditions other than the convenience of deriving the analytical formulae or simplifying the implementation of algorithms.

Thirdly, even though the closed formulae are available for some complex barrier options, this does not necessarily mean that the results are easy to compute. For example, the formulae provided by Heynen and Kat [24] require high-dimensional numerical integration to compute the results.

Fourthly, the numerical methods such as the lattice methods and the Monte Carlo methods often lack generality in the sense that they have to be modified according to different barrier option models and the computation of Greeks can be quite onerous if accuracy and speed is required.

1.2 Methodology employed in the thesis

In **Chapter 4** we discuss the quoting convention and the volatility calibration in the FX markets briefly so that we can see that the Black-Scholes' model fails to price FX options. Accordingly, De Col, Gnoatto and Grasselli [12] have recently presented a novel technique to calibrate the multi-currency pairs simultaneously based on a newly developed type of multi-dimensional Heston model. They demonstrated that the conventional method of calibrating multi-currency pairs has difficulties in matching all volatility smiles. This provides us the inspiration to price a multi-windowed barrier option on a cross-currency pair within their general and consistent framework.

For a complex option pricing problem especially in a complicated model, even if such formula can be derived, we can expect that computing the results would be very demanding. Besides, we have shown that lattice methods and Monte Carlo methods are not ideal to solve our problem. Therefore in the thesis, we utilize

a partial differential equation (PDE) approach [57] to price the multi-windowed barrier options.

It can be demonstrated that the PDE approach can be used in different models with minor alternation. We employ the finite element method (FEM) such as in [56], [18], [55] and [52] to solve the PDEs derived from the barrier option pricing problems. Technically both FD and FEM can be applied since the PDEs arising from the financial problems are generally parabolic. However, a comparison of the two methods indicates that FEM has certain advantages when pricing complex exotic options:

a. Finite element method is ideal to handle complex domains. Hence, it can be used to deal with irregular barriers such as in multi-windowed barrier options;

b. The unstructured mesh system of FEM allows the meshes to be refined where high accuracy of solution is required such as around the barrier curves;

c. To guarantee the existence and uniqueness of the solution with FD, the terminal and boundary conditions have to be sufficiently smooth. These constraints can be weaken when we use FEM.

CSIRO has developed a commercial software named Reditus^{TM} to price a suit of exotic options using FEM. In this system, they apply the space-time FEM which deems the time as another dimension just like the asset price. Unlike the conventional FEM, the spatial and temporal domains are simultaneously discretized in this scheme so that an unstructured mesh is built in the space-time domain.

This method has been explained in details by Hughes and Hulbert [26] and they have shown that the method is capable of capturing the discontinuity with respect to time. It is a good approach to deal with the multi-windowed barrier options because the solutions are discontinuous at where the barriers are changing.

We use the conventional FEM instead of the space-time FEM as the complexities of the latter are beyond the scope of the thesis:

a. In the paper of Hughes and Hulbert [26], they developed a non-linear algorithm even for a linear problem to eliminate the oscillation of the computed solution caused by discontinuity. The algorithm is not simple to implement for a more complicated problem;

b. Because of the increasing of dimension, for an option where three or more assets are involved or an option in the multi-dimensional Heston model, the

domain could be four or more dimensional. Generating an adaptive mesh with the Delaunay tessellation on high dimensional is still a challenge today [4].

1.3 Outline of the thesis

The thesis is outlined as follows. **Chapter 2** presents a discussion about our methodology including deriving the PDE formulations from the stochastic differential equations (SDEs) and the existence and uniqueness of the weak solutions for the general parabolic equations. **Chapter 3** compares the numerical results with the analytical results of a European down-and-out single barrier option and provides the numerical results for the multi-windowed barrier options within Black-Scholes' framework. **Chapter 4** compares the numerical results of a vanilla option with the analytical results and prices a multi-windowed barrier option in Heston's model. In **Chapter 5** we employ the scheme based on the multi-dimensional Heston model proposed by Col, Gnoatto and Grasselli [12] to price a multi-windowed barrier option. As regards implementation, we use GMSH [20] to generate the meshes and the GetFEM++ library [42] together with C++ to obtain the numerical results of PDEs. Finally we present the fully working code in **Appendix B**.

Chapter 2

PDE approach and weak solution

As mentioned in **Chapter 1**, we discuss pricing options in three different models in the thesis: a) Black-Scholes' model [5]; b) Heston's stochastic volatility model [23]; c) a multi-dimensional stochastic volatility model proposed by Col, Gnoatto and Grasselli [12].

In Black-Scholes' model, the asset price S is modeled by the stochastic differential equation

$$dS_t = \mu S_t dt + \sigma S_t dW_t^{\mathbb{P}}, \tag{2.1}$$

where μ and σ are constants and $\left(W_t^{\mathbb{P}}\right)_{t\geq 0}$ is a standard Brownian motion under the probability measure $\mathbb{P}$. In option pricing μ is often called the "drift term" and σ the "volatility" term.

In Heston's model, the volatility term is another stochastic process. Hence, the asset price is driven by

$$\begin{aligned} dS_t &= \mu S_t dt + \sqrt{v_t} S_t dW_t^{\mathbb{P}}, \\ dv_t &= \kappa(\theta - v_t)dt + \xi\sqrt{v_t} dZ_t^{\mathbb{P}}, \end{aligned}$$

where correlation ρ between the two Brownian motions $\left(W_t^{\mathbb{P}}\right)_{t\geq 0}$ and $\left(Z_t^{\mathbb{P}}\right)_{t\geq 0}$ is defined as:

$$\rho dt = d\left\langle W^{\mathbb{P}}, Z^{\mathbb{P}}\right\rangle_t, \quad \rho \in [-1, 1].$$

In the multi-dimensional Heston model of Col, Gnoatto and Grasselli, the asset price is related to two or more "volatility" terms and given by

$$\begin{aligned} dS_t &= S_t\left[\mu dt - (\mathbf{a})^T \mathrm{Diag}(\sqrt{\mathbf{v}_t}) d\mathbf{Z}_t\right]; \\ dv_{t;k} &= \kappa_k(\theta_k - v_{t;k})dt + \xi_k\sqrt{v_{t;k}} dW_{t;k}, \quad k = 1, \ldots, d, \end{aligned} \tag{2.2}$$

where $\mathbf{Z}_t$ is a d-dimensional vector Brownian motion with the elements $Z_{t;k}(k = 1, 2, \ldots, d)$ and $\mathrm{Diag}(\sqrt{\mathbf{v}_t})$ denotes the diagonal matrix with the square root of the

element of the d-dimensional vector $\mathbf{v}_t$ on the principle diagonal. The correlations between the Brownian motions $Z_{t;k}$ and $W_{t;k}$ are given by

when $i \neq j$,

$$\begin{cases} 0 = d\langle Z_i, Z_j \rangle_t, \\ 0 = d\langle W_i, W_j \rangle_t, \\ 0 = d\langle Z_i, W_j \rangle_t, \end{cases}$$

and when $i = j = k$,

$$\rho_k dt = d\langle Z_k, W_k \rangle_t.$$

There are two methods to value an option written on such kind of asset. One is "martingale approach" where the value of option is represented by the conditional expectation of discounted payoff under an equivalent martingale measure. Then the analytic pricing formulae can be derived from this conditional expectation.

The other is "PDE approach" where the value of option is deemed as a function of time t, price S of underlying asset and volatility v if it is not constant and a PDE is derived with this function as the solution. By solving such a PDE, one can obtain the option price.

Under the equivalent martingale measure, the martingale approach and the PDE approach are equivalent with sufficient conditions, i.e, the conditional expectation is the solution of the PDE. The PDE approach allows us to obtain the solution numerically, this is a prominent advantage of the PDE approach.

2.1 PDE and stochastic representation of its solution

Black and Scholes [5] derived their PDE without using the equivalent martingale measure approach. Instead they assume that there are no *arbitrage possibilities* for the self-financed portfolio constructed with a bond and a risky asset.

Definition 2.1. An **arbitrage possibility** exists in a financial market if there is a self-financed portfolio h such that the portfolio value U^h satisfies

$$\begin{aligned} U^h(0) &= 0, \\ \mathbb{P}\left[U^h(T) \geq 0\right] &= 1, \\ \mathbb{P}\left[U^h(T) > 0\right] &> 0, \end{aligned}$$

where $T > 0$. We say that the market is **arbitrage free** if there are no arbitrage possibilities.

Based on this assumption, the PDE with respect to an option value $U = U(S,t)$ is given by

$$\frac{\partial U}{\partial t} + rS\frac{\partial U}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 U}{\partial S^2} - rU = 0 \tag{2.3}$$

where r is the risk-free rate. We can see that the drift term μ in the SDE (2.1) happens to drop out in the derivation of the PDE (2.3).

Unfortunately, this is not the case of Heston's model. Heston [23] derived the PDE with respect to an option value $U = U(S,v,t)$ with the same standard arbitrage free argument like Black and Scholes as

$$\begin{aligned}\frac{\partial U}{\partial t} + rS\frac{\partial U}{\partial S} + \left[\kappa(\theta - v_t) - \lambda(S,v,t)\right]\frac{\partial U}{\partial v} \\ + \frac{1}{2}vS^2\frac{\partial^2 S}{\partial S^2} + \rho\xi vS\frac{\partial^2 U}{\partial v \partial S} + \frac{1}{2}\xi^2 v\frac{\partial^2 U}{\partial v^2} - rU = 0.\end{aligned} \tag{2.4}$$

The unspecified term $\lambda(S,v,t)$ in (2.4) is introduced as the price of volatility risk. Consequently, questions arise such as how to choose the form of $\lambda(S,v,t)$ and how to calibrate the coefficients accordingly.

Although Heston used $\lambda(S,v,t) = \lambda v$ as the form of the volatility risk premium, Kimmel and Aït-Sahalia [30] have shown that it is difficult to choose a proxy to represent the market volatility risk in reality. Hence, we choose an equivalent martingale measure $\mathbb{Q}$ where the discount asset price $e^{-rt}S_t$ is a martingale, which is called "risk-neutral" measure, such that the option price can be valued without calibrating λ.

For example, for (2.1), by applying Itô's formula (Theorem A.1), we get

$$de^{-rt}S_t = e^{-rt}S_t\left[(\mu - r)dt + \sigma dW_t^{\mathbb{P}}\right].$$

Then by Girsanov's Theorem (Theorem A.2) there exists a measure $\mathbb{Q}$ which is equivalent to $\mathbb{P}$ such that

$$W_t^{\mathbb{Q}} = W_t^{\mathbb{P}} + \int_0^t \frac{\mu - r}{\sigma} ds$$

is a Brownian motion under $\mathbb{Q}$ and the Radon-Nikodym derivative is

$$\frac{d\mathbb{Q}}{d\mathbb{P}} = \exp\left[-\int_0^T \frac{\mu - r}{\sigma} dW_t^{\mathbb{P}} - \frac{1}{2}\int_0^T \left(\frac{\mu - r}{\sigma}\right)^2 dt\right].$$

Hence, the asset price S in Black-Scholes' model under the risk-neutral measure $\mathbb{Q}$ is

$$S_t = rS_t dt + \sigma S_t dW_t^{\mathbb{Q}}. \tag{2.5}$$

Similarly, we find a risk-neutral $\bar{\mathbb{Q}}$ for the asset price S in the Heston model such that

$$dS_t = rS_t dt + \sqrt{v_t} S_t dW_t^{\bar{\mathbb{Q}}},$$
$$dv_t = \bar{\kappa}(\bar{\theta} - v_t)dt + \xi\sqrt{v_t} dZ_t^{\bar{\mathbb{Q}}},.$$

where

$$\bar{\kappa} = \kappa + \lambda\xi, \qquad \bar{\theta} = \frac{\kappa\theta}{(\kappa + \lambda\xi)},$$

and

$$\rho dt = d\left\langle W^{\bar{\mathbb{Q}}}, Z^{\bar{\mathbb{Q}}} \right\rangle_t.$$

The corresponding Radon-Nikodym derivative is

$$\frac{d\bar{\mathbb{Q}}}{d\mathbb{P}} = \exp\left(-\int_0^T \lambda\sqrt{v_t} dW_t^{\mathbb{P}} - \frac{1}{2}\int_0^T \lambda^2 v_t dt\right). \tag{2.6}$$

Then (2.4) can be rewritten as

$$\frac{\partial U}{\partial t} + rS\frac{\partial U}{\partial S} + \bar{\kappa}(\bar{\theta} - v_t)\frac{\partial U}{\partial v} + \frac{1}{2}vS^2\frac{\partial^2 S}{\partial S^2} + \rho\xi vS\frac{\partial^2 U}{\partial v \partial S} + \frac{1}{2}\xi^2 v\frac{\partial^2 U}{\partial v^2} - rU = 0. \tag{2.7}$$

Note that till now, the change of measure and the PDEs (2.4) and (2.7) are only valid when the Feller's condition $2\kappa\theta \geq \xi^2 > 0$ is satisfied.

Under a general risk-neutral measure $\mathbb{Q}$, the option price U is given by

$$U = E_{\mathbb{Q}}\left[e^{-r(T-t)} U_T | S_t\right], \tag{2.8}$$

where U_T is the pay-off at the maturity time T. It is supposed to be the stochastic representation of the solution for (2.3) and (2.7), i.e., Feynman-Kac's formula (Theorem A.7) holds for both two models. Heath and Schweizer [22] gives a rigorous proof and sufficient conditions for this proposition and they require the Feller's condition to be satisfied in the Heston model as well.

However, we are concerned with the situation where the Feller's condition is violated in this thesis. This happens commonly in the FX markets, which is discussed in [10] and also can be seen from the calibration results with the real market data in [12]. When $0 < 2\kappa\theta < \xi^2$, the volatility v_t can be 0 before the maturity time T.

First of all, Proposition 2 of [12] states that the Radon-Nikodym derivative (2.6) is a true martingale. Hence, the Girsanov theorem is applied correctly to change the measure $\mathbb{P}$ to the risk-neutral measure $\mathbb{Q}$.

Secondly, the PDE (2.7) is a degenerate parabolic equation when the Feller's condition is violated. Is (2.8) still the stochastic representation of the solution? The existence and uniqueness of the solution in the weighted Hölder space for degenerate parabolic equations has been discussed thoroughly in some literature, e.g., [39]. Recently, in [16], it proves that (2.7) still admits a unique classic solution which is exactly (2.8).

There are several points worth remarking about [16],

i. Heston's model is discussed only (i.e., high dimensional case are not included). However, all the theorems and conditions can be applied directly for the multi-dimensional Heston model in the form of (2.2).

ii. A classic solution for (2.7) exists under the assumption of a continuous terminal condition and Dirichlet boundary conditions. For barrier options, the terminal condition can be discontinuous around the boundary. As demonstrated in [54], the discontinuity will not be propagated into the space-time domain for the parabolic equations if the number of discontinuous points is finite. Therefore, it is safe to conclude that the solution of (2.7) with the discontinuous terminal condition would maintain the same regularity except for $\{T\} \times \partial D$.

iii. There is further work ongoing where "an extension of main results of this article to a broader class of degenerate Markov processes in higher dimensions and more general boundary conditions (including Neumann and oblique boundary conditions)" [16] will be developed [1]. Actually in **Chapter 4**, we can see that the Dirichlet boundary conditions and the Neumann boundary conditions give similar solutions in the Heston model.

Feynman-Kac's Formula (Theorem A.7) is valid in a more general sense for these three models that we discuss in this thesis. The PDE with respect to the option price U has the backward form

$$\frac{\partial U}{\partial t} + \mathbf{V} \cdot \nabla U + \mathbf{M}\nabla \cdot \nabla U - rU = 0,$$

where $\mathbf{V}$ is a n-dimentional vector and $\mathbf{M}$ is a $n \times n$ non-negative symmetric matrix with $n = 1, 2, \cdots$. For the sake of applying numerical methods, we change the variable $\tau = T - t$ such that the PDE is in the form of

$$\frac{\partial U}{\partial \tau} - \mathbf{V} \cdot \nabla U - \mathbf{M}\nabla \cdot \nabla U + rU = 0. \tag{2.9}$$

[1] Prof. Feehan also confirmed personally that they have considered extending the results to other boundary conditions and have some work in progress in a private communication.

In summary, we have illustrated that there exists a unique classic solution for (2.9) which is represented by (2.8). Together with different boundary/initial conditions, the PDE (2.9) can be used to price different type of options. If there exists a unique weak solution, this weak solution must be the same as the classic solution. Consequently we obtain the value of the options by solving (2.9) with a numerical method, e.g., FEM. In next section, we demonstrate the existence and uniqueness of the weak solution.

2.2 Existence and uniqueness of weak solution

We follow [14] to prove the existence and uniqueness of the weak solution to (2.9). Assume D to be an open, bounded subset of $\mathbb{R}^n$ and set $D_T = D \times (0, T]$ for some fixed time $T > 0$.

Notation 2.2. Let $H^k := W^{k,2}(D)$ denote a Sobolev Space, the space $H_0^k := W_0^{k,2}(D)$ is the closure of $C_c^\infty(D)$ in the space $W^{k,2}(D)$ and $H^*(D)$ is the dual space of the space $H_0^1(D)$.

We consider the initial/boundary-value problem

$$\begin{cases} u_t + Lu = f & \text{in } D_T, \\ u = 0 & \text{on } \partial D \times (0, T], \\ u = g & \text{on } D \times (t = 0), \end{cases} \tag{2.10}$$

where $f : D_T \to \mathbb{R}$ and $g : D \to \mathbb{R}$ is given and $u : \bar{D}_T \to \mathbb{R}$ is the unknown with $u = u(x, t)$. The letter L denote a second-order partial differential operator for each time t, having either the divergence form

$$Lu = -\sum_{i,j=1}^{n} \left(a^{ij}(x,t)u_{x_i}\right)_{x_j} + \sum_{i=1}^{n} b^i(x,t)u_{x_i} + c(x,t)u$$

or the non-divergence form

$$Lu = -\sum_{i,j=1}^{n} a^{ij}(x,t)u_{x_i x_j} + \sum_{i=1}^{n} b^i(x,t)u_{x_i} + c(x,t)u.$$

Definition 2.3. If for all $(x, t) \in D_T$, $\xi \in \mathbb{R}^n$, there exists a constant θ such that

$$\sum_{i,j=1}^{n} a^{ij}(x,t)\xi_i\xi_j \geq \theta|\xi|^2,$$

then the operator $\frac{\partial}{\partial t} + L$ is *parabolic*.

For now we assume that

$$a^{ij}, b^i, c \in L^\infty(D_T) \quad (i = 1, \dots, n),$$
$$f \in L^2(D_T),$$
$$g \in L^2(D),$$

and the PDE in (2.10) is a parabolic equation of the divergence form. We always suppose that $a^{ij} = a^{ji}$ for $(i, j = 1, \dots, n)$.

Notation 2.4. We write the time-dependent bilinear form

$$B[u, v; t] = \int_D \left[\sum_{i,j=1}^{n} a^{ij}(\cdot, t) u_{x_i} v_{x_j} + \sum_{i=1}^{n} b^i(\cdot, t) u_{x_i} v + c(\cdot, t) uv \right] dx$$

for $u, v \in H_0^1(D)$ and a.e. $0 \le t \le T$.

Now we consider u not as a function of x and t together, but as a mapping $\tilde{u}$ of t in to the space $H_0^1(D)$ of x, i.e.,

$$\tilde{u} : [0, T] \to H_0^1(D)$$

defined by

$$[\tilde{u}(t)](x) := u(x, t) \quad (x \in D, 0 \le t \le T).$$

Thus, in the problem (2.10), similarly we define

$$\tilde{f} : [0, T] \to L^2(D)$$

by

$$[\tilde{f}(t)](u) := f(x, t) \quad (x \in D, 0 \le t \le T)$$

Then, if we fix function $v \in H_0^1$, we can multiply the PDE $\frac{\partial u}{\partial t} + Lu = f$ by v and integrate by parts to find

$$(\dot{\tilde{u}}, v) + B[\tilde{u}, v; t] = (\tilde{f}, v)$$

for each $0 \le t \le T$, where $(\cdot, \cdot)$ denotes the inner product in $L^2(D)$ and $\dot{u} = \frac{du}{dt}$. Finally, by the Theorem A.3, we can see that $\dot{\tilde{u}} \in H^*(D)$.

Next, we define the *weak solution* of a parabolic equation as follow

Definition 2.5. A function

$$\tilde{u} \in L^2(0, T; H_0^1(D)) \quad \text{with } \dot{\tilde{u}} \in L^2(0, T; H^*(D))$$

is a *weak solution* of the parabolic initial/boundary-value problem (2.10) provided

(i) $(\dot{\tilde{u}}, v) + B[\tilde{u}, v; t] = (\tilde{f}, v)$ for each $v \in H_0^1(D)$ and a.e. time $0 \le t \le T$,

(ii) $\tilde{u}(0) = g$.

We apply Galerkin's method to obtain the solution of the problem (2.10). Suppose that the functions $w_k := w_k(x)$ $(k = 1, \ldots)$ are smooth,

$$\{w_k\}_{k=1}^{\infty} \text{ is an orthogonal basis of } H_0^1(D)$$

and

$$\{w_k\}_{k=1}^{\infty} \text{ is an orthonormal basis of } L^2(D)$$

Now we look for a function $\tilde{u}_m : [0, T] \to H_0^1(D)$ of the form

$$\tilde{u}_m(t) := \sum_{k=1}^{m} d_m^k w_k, \tag{2.11}$$

where the coefficients $d_m^k(t)$ $(0 \le t \le T, k = 1, \ldots, m)$ satisfy

$$d_m^k(0) = (g, w_k) \quad (k = 1, \ldots, m) \tag{2.12}$$

and

$$(\dot{\tilde{u}}_m, w_k) + B[\tilde{u}_m, w_k; t] = (\tilde{f}, w_k) \quad (0 \le t \le T, k = 1, \ldots, m) \tag{2.13}$$

for a fixed positive integer m.

Thus, first of all, we seek a function u_m in a finite-dimensional subspace spanned by $\{w_k\}_{k=1}^m$. Next we let $m \to \infty$ to build a weak solution of the problem (2.10).

Theorem 2.6. *For each integer $m = 1, 2, \ldots$ there exists a unique function u_m with the form (2.11) satisfying (2.12) and (2.13)*

Proof. Assume $\tilde{u}_m$ has the structure (2.11), we can get

$$\left(\dot{\tilde{u}}_m(t), w_k\right) = \dot{d}_m^k(t)$$

and

$$B[\tilde{u}_m, w_k; t] = \sum_{l=1}^{m} e_{kl}(t) d_m^l(t),$$

where $e_{kl}(t) := B[w_l, w_k; t]$ $(k, l = 1, \ldots, m)$. Let $f^k(t) = (\tilde{f}(t), w_k)$ $(k = 1, \ldots, m)$. Then (2.13) becomes the linear system of ordinary differential equation (ODE)

$$\dot{d}_m^k(t) + \sum_{l=1}^{m} e_{kl}(t) d_m^l(t) = f^k(t) \quad (k = 1, \ldots, m)$$

subject to the initial condition (2.12). According to the standard existence theory for ODEs (see [11]), there exists a unique solution $\tilde{d}_m(t) = (d_m^1(t), \ldots, d_m^m(t))$ for the ODE for a.e. $0 \le t \le T$. Hence $\tilde{u}_m$ defined by (2.11) solves (2.13) for a.e. $0 \le t \le T$. □

To prove the uniqueness of the weak solution, we need the following theorem:

Theorem 2.7. *There exists a constant C, depending only on D, T and the coefficients of L but not on m, such that*

$$\max_{0\le t\le T} \|\tilde{u}_m(t)\|_{L^2(D)} + \|\tilde{u}_m\|_{L^2(0,T;H_0^1(D))} + \|\dot{\tilde{u}}_m\|_{L^2(0,T;H^*(D))} \le C\left(\|\tilde{f}\|_{L^2(0,T;L^2(D))} + \|g\|_{L^2(D)}\right)$$

The details of the proof of the above theorem, we refer the reader to [14].

Theorem 2.8. *There exists a unique solution of the problem (2.10).*

Proof. According to Theorem 2.7, we can see that the sequence $\{\tilde{u}_m\}_{m=1}^{\infty}$ is bounded in $L^2(0,T;H_0^1(D))$ and $\{\dot{\tilde{u}}_m\}_{m=1}^{\infty}$ is bounded in $L^2(0,T;H^*(D))$.

Thus, there exists a subsequence $\{\tilde{u}_{m_l}\}$ and a function $\tilde{u} \in L^2(0,T;H_0^1(D))$ with $\dot{\tilde{u}} \in L^2(0,T;H^*(D))$, such that

$$\begin{cases} \tilde{u}_{m_l} \to \tilde{u} & \text{weakly in } L^2(0,T;H_0^1(D)), \\ \dot{\tilde{u}}_{m_l} \to \dot{\tilde{u}} & \text{weakly in } L^2(0,T;H^*(D)). \end{cases}$$

We fix an integer N and choose a function $\tilde{v} \in C^1(0,T;H_0^1(D))$ having the form

$$\tilde{v}(t) = \sum_{k=1}^{N} d^k(t) w_k, \tag{2.14}$$

where $\{d^k\}_{k=1}^N$ are given functions. We choose $m \ge N$, multiply (2.13) by $d^k(t)$ and then integrate with respect to t

$$\int_0^T (\dot{\tilde{u}}_m, \tilde{v}) + B[\tilde{u}_m, \tilde{v}; t]dt = \int_0^T (\tilde{f}, \tilde{v})dt. \tag{2.15}$$

Let $m = m_l \to \infty$, we obtain

$$\int_0^T (\dot{\tilde{u}}, \tilde{v}) + B[\tilde{u}, \tilde{v}; t]dt = \int_0^T (\tilde{f}, \tilde{v})dt.$$

The equality holds for all functions $\tilde{v} \in L^2(0,T;H_0^1(D))$ since the functions of form (2.14) are dense in this space. Hence, we have

$$(\dot{\tilde{u}}, v) + B[\tilde{u}, v; t] = (\tilde{f}, v),$$

for each $v \in H_0^1(D)$ and a.e. $0 \le t \le T$.

For each $\tilde{v} \in C^1(0,T;H_0^1(D))$ with $\tilde{v}(T) = 0$, we have

$$\int_0^T -(\dot{\tilde{v}}, \tilde{u}) + B[\tilde{u}, \tilde{v}; t] = \int_0^T (\tilde{f}, \tilde{v})dt + (\tilde{u}(0), \tilde{v}(0)).$$

And from (2.15), we deduce

$$\int_0^T -(\dot{\tilde{v}}, \tilde{u}_m) + B[\tilde{u}_m, \tilde{v}; t] = \int_0^T (\tilde{f}, \tilde{v})dt + (\tilde{u}_m(0), \tilde{v}(0)).$$

By taking $m \to \infty$, we have

$$\int_0^T -(\dot{\tilde{v}}, \tilde{u}) + B[\tilde{u}, \tilde{v}; t] = \int_0^T (\tilde{f}, \tilde{v})dt + (g, \tilde{v}(0)).$$

As $\tilde{v}(0)$ is arbitrary, we conclude $\tilde{u}(0) = g$.

To prove the uniqueness of the solution, it is sufficient to prove

$$\tilde{u} \equiv 0$$

when $\tilde{f} \equiv g \equiv 0$. When $\tilde{u} = \tilde{v}$ and $\tilde{f} = 0$, we have

$$(\dot{\tilde{u}}, \tilde{u}) + B[\tilde{u}, \tilde{u}; t] = 0.$$

After applying (ii) of Theorem A.4, we have

$$\frac{d}{dt}\left(\frac{1}{2}\|\tilde{u}\|^2_{L^2(D)}\right) + B[\tilde{u}, \tilde{u}; t] = 0.$$

Since according to Theorem A.5, we have the inequality

$$B[\tilde{u}, \tilde{u}; t] \ge \beta\|\tilde{u}\|^2_{H_0^1(D)} - \gamma\|\tilde{u}\|^2_{L^2(D)} \le -\gamma\|\tilde{u}\|^2_{L^2(D)},$$

Gronwall's inequality (Theorem A.6) implies $\tilde{u} \equiv 0$. □

2.3 Finite element method and error estimates

The PDE (2.9) is of non-divergence form, so firstly we transfer it to the divergence form

$$\frac{\partial U}{\partial \tau} - \widehat{\mathbf{V}} \cdot \nabla U - \nabla \cdot \widehat{\mathbf{M}} \nabla U + rU = 0, \tag{2.16}$$

so that the weak formulation can be easily generated.

Assume the domain D is a bounded Lipschitz domain since the computational domain for our problems is square (2-dimensional) or cubic (3-dimensional). We suppose that Γ_1 and Γ_2 form a disjoint decomposition of the boundary ∂D:

$$\partial D = \Gamma_1 \cup \Gamma_2,$$

where Γ_2 is a closed subset of the boundary ∂D. Then the boundary conditions are:

$$\widehat{\mathbf{M}}\nabla U \cdot \vec{\mathbf{n}} = g_1 \qquad \text{on } \Gamma_1 \times (0, T],$$

where $\vec{\mathbf{n}}$ is the outward normal vector of the boundary sphere Γ_1, and

$$U = g_2 \qquad \text{on } \Gamma_2 \times (0, T]. \tag{2.17}$$

Therefore the weak formulation is

$$\int_D \dot{U} v dD - \int_D \widehat{\mathbf{V}} \cdot \nabla U v dD + \int_D \widehat{\mathbf{M}}\nabla U \cdot \nabla v dD + r \int_D U v dD = \int_{\Gamma_1} g_1 v dS := b(v; \tau) \tag{2.18}$$

for all $v \in V$, where V denotes the function space and S denotes the boundary sphere. The problem is to find the solution numerically so that it satisfies (2.18) and (2.17) and has the value of U_0 when $\tau = 0$.

2.3.1 Finite element method

In order to solve above problem, we use a semi-discretization scheme as follow:

i Discretise the space domain and apply Galerkin's method to obtain an ODE of the unknown:

First, a partition $\mathcal{T}_h$ of the domain D is generated by closed triangles (2-dimensional) or tetrahedrons (3-dimensional) K (i.e., including the boundary ∂K) with the following properties:

a) $\overline{D} = \bigcup_{K \in \mathcal{T}_h} K$;

b) For $K, K' \in \mathcal{T}_h$,

$$\text{int}(K) \cap \text{int}(K') = \emptyset$$

where $\text{int}(K) := K \backslash \partial K$;

c) If $K \neq K'$ but $K \cap K' \neq \emptyset$, then $K \cap K'$ is either a point or a common edge of K and K'.

The subscript h is the maximum length of the edges of all the triangles or tetrahedrons.

Then we choose a Lagrange basis $\{w_k\}_{k=1}^{m_h}$ on $\mathcal{T}_h$ such that the weak formulation of (2.16) on a function space V_h spanned by $\{w_k\}_{k=1}^{m_h}$ is

$$\int_D \dot{U}_h v_h dD - \int_D \widehat{\mathbf{V}} \cdot \nabla U_h v_h dD + \int_D \widehat{\mathbf{M}} \nabla U_h \cdot \nabla v_h dD + r \int_D U_h v_h dD = b(v_h; \tau), \quad (2.19)$$

for all $v_h \in V_h$.

Since $U_h = \sum_{j=1}^{m_h} w_j$ and $v_h = \sum_{i=1}^{m_h} w_i$, finally we obtain the ODE as follow:

$$\sum_{j=1}^{m_h} \dot{u}_j \int_D w_j w_i dD + \sum_{j=1}^{m_h} u_j \left[- \int_D \widehat{\mathbf{V}} \cdot \nabla w_j w_i + \int_D \widehat{\mathbf{M}} \nabla w_j \cdot \nabla w_i dD + r \int_D w_j w_i dD \right] = b(w_i; \tau), \quad (2.20)$$

together with the initial condition $U_{0;h} = \sum_{j=1}^{m_h} u_{0;j} w_j$ for $i = 1, \ldots, m_h$, where $\mathbf{u} = (u_1, \ldots, u_{m_h})^T$ is the unknown vector.

ii Utilize the backward Euler method, a type of finite difference method, on the time domain:

We obtain the linear system with respect to $\mathbf{u}$ from (2.20) in the form of

$$\mathbf{A} \frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} + \mathbf{B}\mathbf{u}^{n+1} = \mathbf{q} \quad (2.21)$$

where $\mathbf{u}^n$ is the solution of unknown vector in the time step n.

iii Solve the linear system (2.21) in each time step to obtain the solution.

2.3.2 Error estimates

Let V_h be a finite subspace of V. The weak solution $\tilde{U}$ of (2.18) is a function with $\tilde{U} \in L^2(0, T; V(D))$ and $\dot{\tilde{U}} \in L^2(0, T; L^2(D))$, and the weak solution $\tilde{U}_h$ of (2.20) is a function with $\tilde{U}_h \in L^2(0, T; V_h(D))$ and $\dot{\tilde{U}}_h \in L^2(0, T; L^2(D))$.

We introduce the elliptic projection of the solution $\tilde{U}$ in error estimates.

Definition 2.9. The elliptic projection $R_h : V \to V_h$ is defined by

$$u \mapsto R_h u \quad \Longleftrightarrow \quad B[R_h u - u, u_h; t] = 0 \quad \forall\, u_h \in V_h.$$

Suppose the bilinear form $B[u, v; t]$ is continuous and positive definite, the elliptic projection R_h has the following properties

Theorem 2.10. *Under the assumption of Definition 2.9:*

(i) R_h *is linear and continuous;*

(ii) R_h *yields quasi-optimal approximations given by*

$$\|u - R_h u\|_V \leq C \inf_{u_h \in V_h} \|u - u_h\|_V$$

where C *does not depend on* u *or* h.

As shown in last section, to admit a unique solution for the parabolic equations, the bilinear form B does not have to be positive definite, but satisfies the condition *(ii)* in Theorem A.5. However, the conditions which the elliptic projection R_h requires can be satisfied by changing the variable $\widehat{u} = e^{-\gamma t}u$ and we define a new bilinear form as

$$B_\gamma[v, v; t] = B[v, v; t] + \gamma\|v\|_{L^2}.$$

Then (2.18) can be written as

$$(\dot{\widehat{U}}, v) + B_\gamma[\widehat{U}, v; \tau] = e^{-\gamma\tau} b(v; \tau)$$

and (2.19) as

$$(\dot{\widehat{U}}_h, v_h) + B_\gamma[\widehat{U}_h, v_h; \tau] = e^{-\gamma\tau} b(v_h; \tau)$$

Now we have the following result as the semi-discrete error estimate in L^2 norm:

Theorem 2.11. *Suppose* $U_0 \in V$ *and* $U_{0;h} \in V_h$. *Then if* $\tilde{U}(\tau)$ *is sufficiently smooth,*

$$\begin{aligned}\|\tilde{U}(\tau) - \tilde{U}_h(\tau)\|_{L^2} \leq & \|U_{0;h} - R_h U_0\|_{L^2} e^{-\beta\tau} \\ & + \|\tilde{U}(\tau) - R_h\tilde{U}(\tau)\|_{L^2} + \int_0^\tau \|\dot{\tilde{U}}(s) - R_h\dot{\tilde{U}}(s)\|_{L^2} e^{-\beta(\tau-s)} ds.\end{aligned}$$

To prove this theorem, we need the result from the following lemma:

Lemma 2.12. *Let* B *be a positive definite, bilinear form,* $u_0 \in L^2(D)$ *and suppose the considered boundary conditions are homogeneous. Then, for the weak solution* $\tilde{u}$ *of (2.10) the following estimate holds:*

$$\|\tilde{u}(t)\|_{L^2} \leq \|u_0\|_{L^2} e^{-\beta\tau} + \| \int_0^t \tilde{f}(s)\|_{L^2} e^{-\beta(\tau-s)} ds.$$

The proof of Lemma 2.12 can be found in [32].

Proof of Theorem 2.11. The error can be decomposed as

$$\tilde{U}_h(\tau) - \tilde{U}(\tau) = \tilde{U}_h(\tau) - R_h\tilde{U}(\tau) + R_h\tilde{U}(\tau) - \tilde{U}(\tau) := \theta(\tau) + \delta(\tau).$$

Then we take $v = v_h \in V_h$ in (2.18) to obtain

$$\left(\dot{\tilde{U}}(\tau), v_h\right) + B\left[\tilde{U}(\tau), v_h; \tau\right] = \left(\dot{\tilde{U}}(\tau), v_h\right) + B\left[R_h\tilde{U}(\tau), v_h; \tau\right] = b(v_h; \tau).$$

After subtracting this equation from (2.19), we have

$$\left(\dot{\tilde{U}}_h(\tau), v_h\right) - \left(\dot{\tilde{U}}(\tau), v_h\right) + B\left[\theta(\tau), v_h; \tau\right] = 0.$$

Thus

$$\left(\dot{\theta}(\tau), v_h\right) + B\left[\theta(\tau), v_h; \tau\right] = \left(\dot{\tilde{U}}(\tau), v_h\right) - \left(\dot{R}_h\tilde{U}(\tau), v_h\right) = -\left(\dot{\delta}(\tau), v_h\right)$$

Then we apply Lemma 2.12 to get

$$\|\theta(\tau)\|_{L^2} \le \|\theta(0)\|_{L^2}e^{-\beta\tau} + \int_0^\tau \|\dot{\delta}(s)\|_{L^2}e^{-\beta(\tau-s)}ds.$$

Since R_h is continuous and $\tilde{U}(\tau)$ is sufficiently smooth, we have $\dot{\delta}(\tau) = \dot{\tilde{U}}(\tau) - R_h\dot{\tilde{U}}(\tau)$. Hence, we have the estimate as stated by the theorem. □

The error norm in Theorem 2.11 is estimated by

i the initial error, which occurs only if $U_{0;h}$ does not coincide with the elliptic projection of U_0;

ii the projection error of the exact solution measured in the norm of L^2;

iii the projection error of $\dot{\tilde{U}}(\tau)$ measured in the norm of L^2 and integrally weighted by $e^{-\beta(\tau-s)}$on $(0, \tau)$.

If $U_0 \in V \cap H^2(D)$ and $\tilde{U}(t) \in V \cap H^2(D)$, we have an optimal L^2-error estimate.

Theorem 2.13. *Let the space $V_h \in V$ be such that for any function $w \in V \cap H^2(D)$,*

$$\inf_{v_h \in V_h} \|w - v_h\|_V \le Ch|w|_2,$$

where the constant $C > 0$ does not depend on h and w. We have

$$\|\tilde{U}(\tau) - \tilde{U}_h(\tau)\|_{L^2} \le \|U_{0;h} - U_0\|_{L^2}e^{-\beta\tau} + Ch^2\left(\|U_0\|_{H^2} + \|\tilde{U}(\tau)\|_{H^2} + \int_0^\tau \|\dot{\tilde{U}}(s)\|_{H^2}e^{-\beta(\tau-s)}ds\right).$$

This result is directly from the elliptic projection error

$$\|w - R_h w\|_{L^2} \leq Ch^2 \|w\|_{H^2} \quad \forall\, w \in V \cap H^2(D).$$

Similarly, we obtain the fully-discrete error estimate:

$$\begin{aligned}\|\tilde{U}_h^n - \tilde{U}^n\|_{L^2} \leq & \|U_{0;h} - R_h U_0\|_{L^2} + \|\tilde{U}^n - R_h \tilde{U}^n\|_{L^2} \\ & + \int_0^{\tau_n} \|\dot{\tilde{U}}(s) - R_h \dot{\tilde{U}}(s)\|_{L^2} ds + \Delta\tau \int_0^{\tau_n} \|\ddot{\tilde{U}}(s)\|_{L^2} ds,\end{aligned}$$

when $U_0 \in V$ and $\tilde{U} \in C^2(0, T, V)$, and the optimal error estimate:

$$\begin{aligned}\|\tilde{U}_h^n - \tilde{U}^n\|_{L^2} \leq & \|U_{0;h} - U_0\|_{L^2} + Ch^2 \left(\|U_0\|_{H^2} + \|\tilde{U}^n\|_{H^2} + \int_0^{\tau_n} \|\dot{\tilde{U}}(s)\|_{H^2} ds \right) \\ & + \Delta\tau \int_0^{\tau_n} \|\ddot{\tilde{U}}(s)\|_{L^2} ds\end{aligned}$$

when $U_0 \in V \cap H^2(D)$ and $\tilde{U} \in C^2(0, T, V \cap H^2(D))$. These results are extended to the degenerate parabolic equations in [28].

Chapter 3

Barrier options in Black-Scholes' model

In this chapter, we consider the "classic" case of the Black-Scholes' model. This allows us to recall some well-known results and to demonstrate the robustness of our numerical implementation in a simple setting.

3.1 Constant barrier option

3.1.1 Closed-form formulae

We consider an European down-and-out call option with the constant barrier value c_d, the strike price K and the expiration time T. Let $S_*(t) = \min\{S_u : 0 \leq u \leq t\}$ and $S^*(t) = \max\{S_u : 0 \leq u \leq t\}$ where S_t follows the SDE (2.5). The payoff of this option at time T is

$$C_T = \mathbf{1}_{\{S_*(T) > c_d, S_T > K\}}(S_T - K).$$

The value of such an option at time $t \leq T$ under the risk-neutral measure $\mathbb{Q}$ is

$$U(S,t) = e^{-r(T-t)} E_{\mathbb{Q}}(C_T | S_t). \tag{3.1}$$

Then we can derive the closed-form formula for (3.1) as follows

Proposition 3.1 (Down-and-out call option)**.** *The down-and-out call European option is priced as follows:*

For $c_d < K$

$$U(S,t) = C(t,s,K) - \left(\frac{c_d}{s}\right)^{\frac{2r-\sigma^2}{\sigma^2}} C\left(t, \frac{c_d^2}{s}, K\right),$$

For $c_d > K$

$$U(S,t) = C(t,s,c_d) + (c_d - K)H(t,s,c_d) - \left(\frac{c_d}{s}\right)^{\frac{2r-\sigma^2}{\sigma^2}} \left[C\left(t, \frac{c_d^2}{s}, c_d\right) + (c_d - K)H\left(t, \frac{c_d^2}{s}, c_d\right)\right],$$

where

$$\begin{aligned} C(t,s,K) &= s\Phi\left[d_1(t,s,K)\right] - Ke^{-r(T-t)}\Phi\left[d_2(t,s,K)\right], \\ H(t,s,c_d) &= e^{-r(T-t)}\Phi\left[d_2(t,s,c_d)\right], \\ d_1(t,s,K) &= \frac{\ln\left(\frac{s}{K}\right) + \left(r + \frac{1}{2}\sigma^2\right)(T-t)}{\sigma\sqrt{T-t}}, \\ d_2(t,s,K) &= \frac{\ln\left(\frac{s}{K}\right) + \left(r - \frac{1}{2}\sigma^2\right)(T-t)}{\sigma\sqrt{T-t}}, \end{aligned}$$

and Φ *is cumulative standard normal distribution function.*

3.1.2 Numerical solution

Recall that the price of the same down-and-out barrier option in Proposition 3.1, i.e., $U = U(S,\tau)$ where $\tau = T - t$, follows the backward equation

$$\frac{\partial U}{\partial \tau} - rS\frac{\partial U}{\partial S} - \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 U}{\partial S^2} + rU = 0, \tag{3.2}$$

with the boundary/initial conditions

$$\begin{cases} U(S,0) = \max(S-K,0), \\ U(S,\tau) = 0 \quad \text{when } S = c_d,\ \tau \in [0,T], \\ U(S,\tau) = S \quad \text{when } S \to \infty,\ \tau \in [0,T]. \end{cases} \tag{3.3}$$

In the implementation, we need to deal with the boundary condition when $S \to \infty$ with caution. First of all, we take S to be relatively large, e.g., 8 to 10 times of the strike price K. Secondly, the value of U at the boundary, where we set as $S_{\max} = 8K$, changes with time so that U is a function with respect to the time variable τ given by

$$U(S,t) = S - Ke^{-r\tau} \quad \text{when } S = 8K. \tag{3.4}$$

Next we employ the methodology described in **Chapter 2** to obtain the weak solution of (3.2) with the initial/boundary conditions (3.3). We use the parameter

Parameter	r	σ	c_d	K	T
Value	0.10	0.20	99.9	100.0	0.5

Table 3.1: European down-and-out call parameter values

values as shown in Table 3.1, which are also used in [57] where the finite difference method is applied to price an European down-and-out call option.

We obtain the results in Table 3.2 with a relatively coarse mesh with $\Delta S = 0.5$ near the barrier ($S \in (99.9, 105.0)$) and $\Delta S = 1.0$ otherwise ($S \in (105.0, 800.0)$). The time step is set to $\Delta\tau = 0.05$. The results in Table 3.2 have shown the consistency even when the asset price is very close to the barrier, i,e., $S = 100$. This is one main advantage of the PDE method over the lattice methods. The result of $U = 0.160$ when $S = 100.0$ also matches the numerical result obtained in [57].

S	100.0	102.0	105.0	115.0	150.0	199.0
PDE	0.160	3.256	7.426	19.138	54.620	103.618
Analytic	0.165	3.301	7.537	19.354	54.876	103.877

Table 3.2: European down-and-out call values. The mesh is non-uniform with $\Delta S = 0.5$ when $S \in (99.9, 105)$ and $\Delta S = 1.0$ when $S \in (105.0, 800.0)$. The time step is $\Delta\tau = 0.05$.

The numerical results in Table 3.2 have an accuracy in sup norm

$$\|U_{\text{numerical}} - U_{\text{analytic}}\|_\infty = \begin{cases} 0.1 & \text{when } \Delta S = 0.5, \\ 0.3 & \text{when } \Delta S = 1.0. \end{cases}$$

This is an acceptable outcome for a coarse mesh like this.

Then we refine the mesh to $\Delta S = 0.1$ when $S \in (99.9, 105.0)$ and $\Delta S = 0.5$ when $S \in (105.0, 800.0)$. The time step is set to $\Delta\tau = 0.025$. The plots of the numerical results and analytic results are shown in Figure 3.1 and the difference between the two types of results is plotted in Figure 3.2.

We can see that the accuracy is very high in Figure 3.2 and also we have the sup norm:

$$\|U_{\text{numerical}} - U_{\text{analytic}}\|_\infty = \begin{cases} 0.1 & \text{when } \Delta S = 0.5, \\ 0.025 & \text{when } \Delta S = 0.1. \end{cases}$$

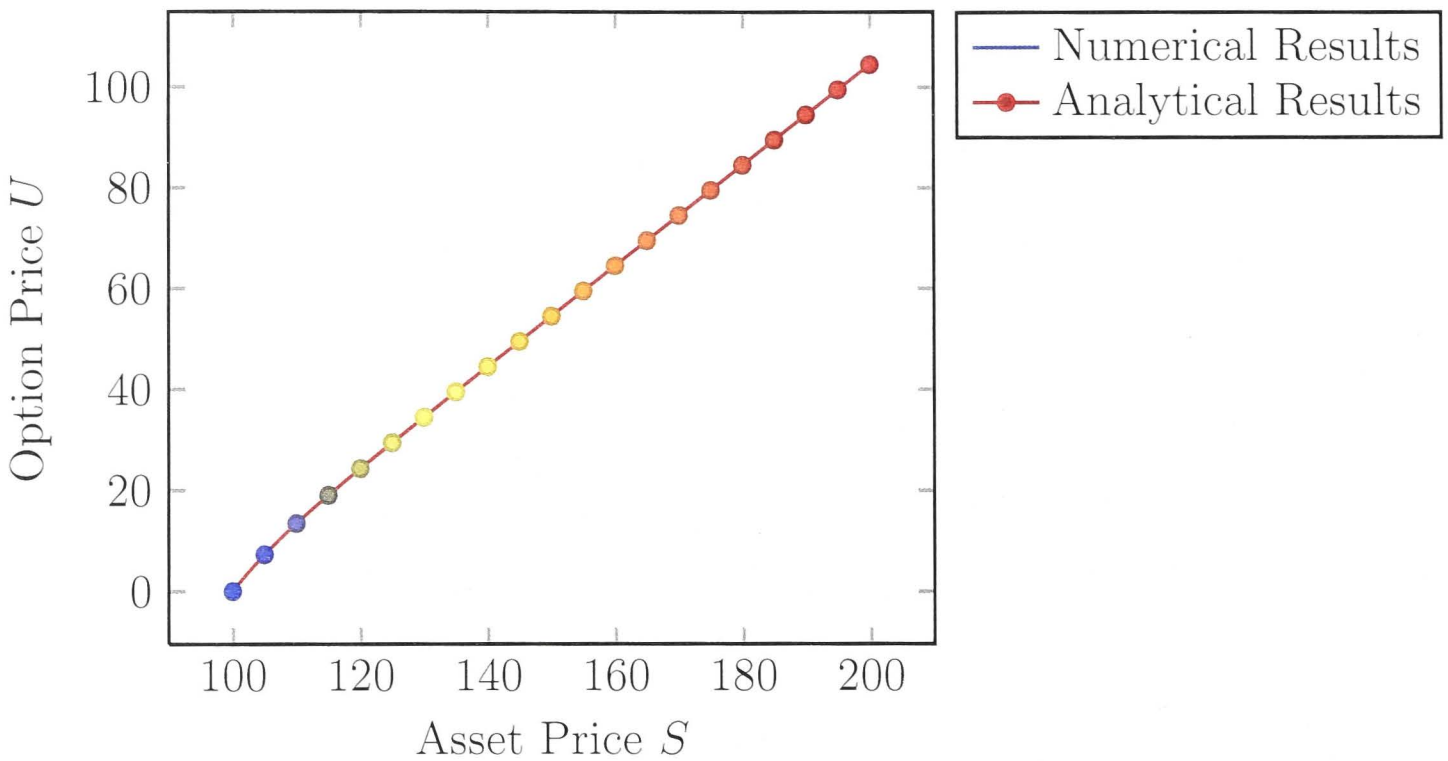

Figure 3.1: European down-and-out call values. The mesh is non-uniform with $\Delta S = 0.1$ *when* $S \in (99.9, 105)$ *and* $\Delta S = 0.5$ *when* $S \in (105.0, 800.0)$*. The time step is* $\Delta\tau = 0.025$*.*

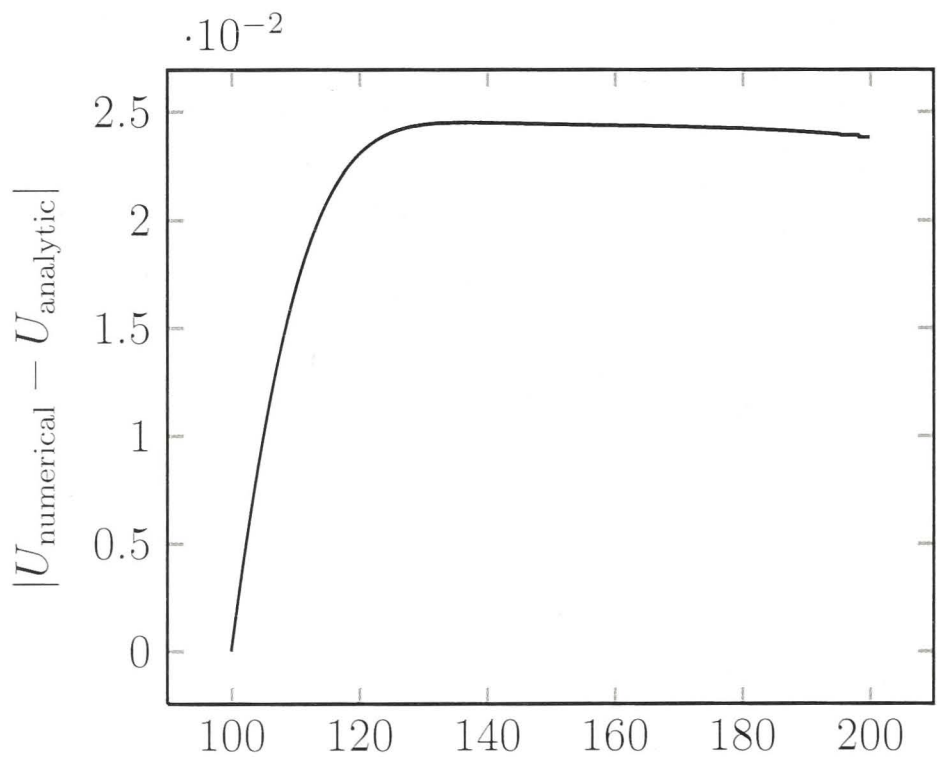

Figure 3.2: Difference between the numerical results and the analytic results. The mesh is non-uniform with $\Delta S = 0.1$ *when* $S \in (99.9, 105)$ *and* $\Delta S = 0.5$ *when* $S \in (105.0, 800.0)$*. Time step is* $\Delta\tau = 0.025$*.*

We can still use a finer mesh with $\Delta S = 0.01$ when $S \in (99.9, 105.0)$ and $\Delta S = 0.05$ when $S \in (105.0, 800.0)$ to improve the accuracy. However, Table 3.3 demonstrates that a fine mesh like this makes the computation much more costly

without improving the results significantly. We can see that only less than 0.003 difference is achieved by refining the mesh by an order of 10. Hence, we use the "fine mesh" in the next section when we price a multi-windowed barrier option.

Mesh		Coarse	Fine	Finer
Δt		0.05	0.025	0.0025
S	100.0	0.160	0.163	0.163
	102.0	3.256	3.292	3.293
	115.0	19.238	19.312	19.314
	150.0	54.620	54.828	54.831
Normalized time cost		1	6.16	765.21

[1] Coarse mesh: $\Delta S = 0.5$ when $S \in (99.9, 105.5)$ and $\Delta S = 1.0$ when $S \in (105.0, 800)$; fine Mesh: $\Delta S = 0.1$ when $S \in (99.9, 105.5)$ and $\Delta S = 0.5$ when $S \in (105.0, 800)$; finer Mesh: $\Delta S = 0.01$ when $S \in (99.9, 105.5)$ and $\Delta S = 0.05$ when $S \in (105.0, 800)$.

[2] The time cost is without the time used to generate the meshes; the normalized time is obtained based on the time used to compute the results with the coarse mesh.

Table 3.3: European down-and-out call values with different meshes.

3.2 Multi-windowed barrier option

We now consider a typical multi-windowed call option which is illustrated in Figure 3.3. It can be seen that the barriers change discretely with time: from $t = 0$ to $t = 0.1$, it is a double knock-out option; from $t = 0.1$ to $t = 0.2$, it is a vanilla option; from $t = 0.2$ to $t = 0.3$, it is a single down-and-out option; then from $t = 0.3$ to $t = 0.5$, it is back to be a double knock-out option again but with the barriers different from previous.

Figure 3.3 shows all possible scenarios during the life of a multi-windowed barrier option. We can see that any multi-windowed barrier option can be deemed as a finite sequence of option A and option B as demonstrated in Figure 3.4 and Figure 3.5 respectively. Hence, if we can value these two options appropriately, we can price any given multi-windowed barrier option.

The complication of pricing a multi-windowed barrier option is that the computational domain changes discretely with time. For example, in Figure 3.4, the space domain is $[c_d, c_u]$ when $t \in [0, T_1]$ and then changes to $[0, \infty)$ when

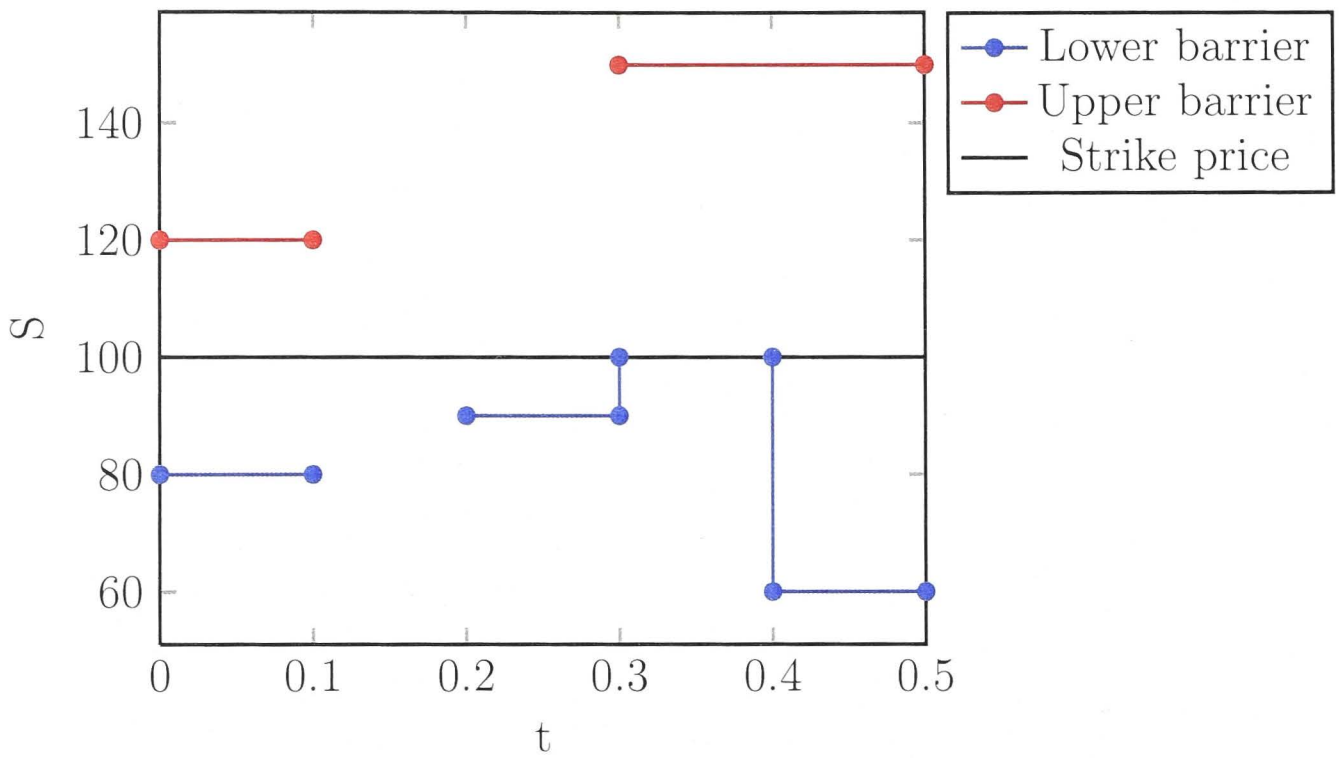

Figure 3.3: A multi-windowed barrier option

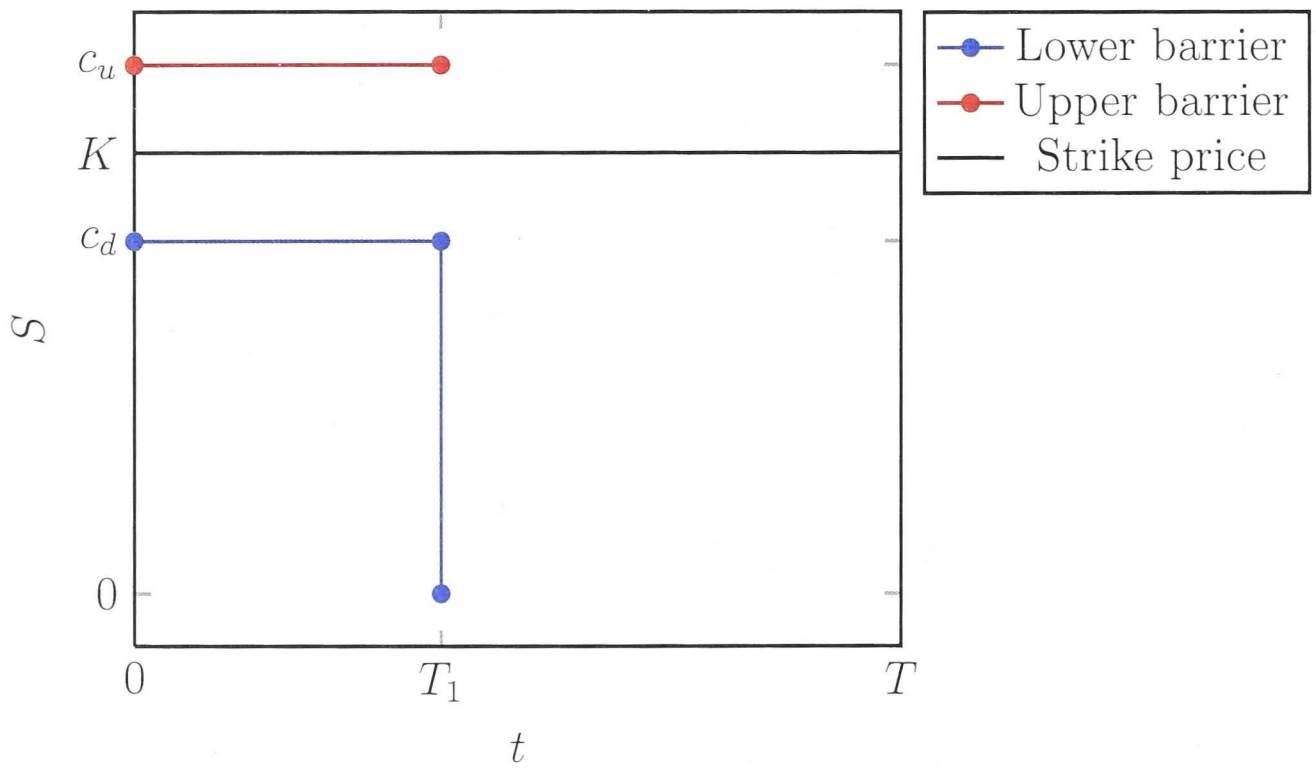

Figure 3.4: Option A: the option has double barriers from time 0 to T_1 and then works as a normal vanilla option until expiration time T.

$t \in (T_1, T]$. In order to apply the conventional FEM, we decompose the problem into two sub-problems based on the time when the barriers change. One is to solve the PDE on the space-time domain $[c_d, c_u] \times [0, T_1]$ and the other is to solve the PDE on the space-time domain $[0, \infty) \times [T_1, T]$. Thus, we take two steps:

1. Solve the PDE with the terminal/boundary conditions on $[0, \infty) \times [T_1, T]$ to obtain the solution at time T_1;

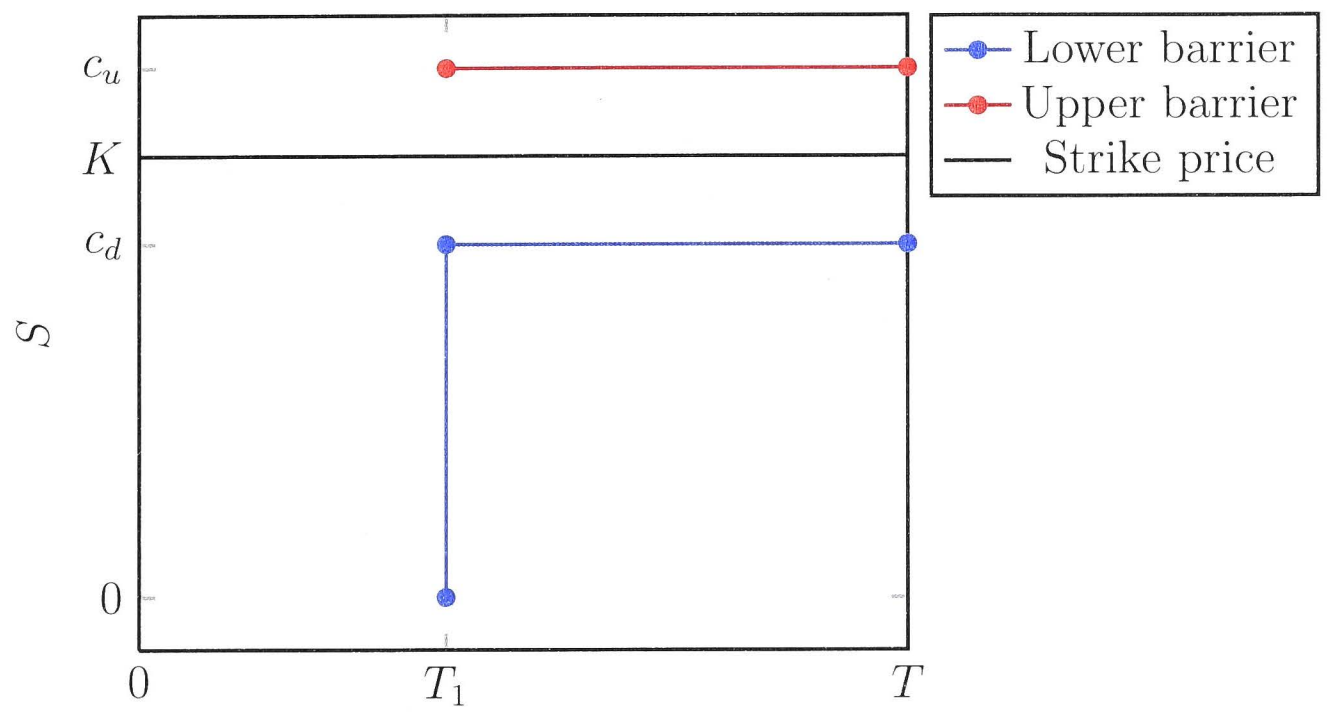

Figure 3.5: Option B: the option works as a vanilla option from time 0 to T_1 and then has double barriers until expiration time T.

2. Interpolate the solution at T_1 on the new space domain $[c_d, c_u]$ to get the new terminal condition and then solve the PDE to obtain the final solution.

While it is the same approach for option B, the terminal/boundary conditions at time T_1 are not as obvious as those of the option A. It is easy to perceive that for option A, the terminal/boundary conditions in $[0, T_1]$ are

$$\begin{cases} U(c_d, t) & = 0, \\ U(c_u, t) & = 0, \\ U(S, T_1) & = U^*(S), \end{cases}$$

where $U^*(S)$ is the solution obtained on step 1. We simply keep the values on the domain (c_d, c_u) and discard the rest. The boundary conditions are obtained by the nature of a double knock-out option, i.e., the option has no value once the price of the underlying asset hits either barrier before the expiration.

For option B, the boundary condition $U(S, t)$ where $t \in [0, T_1)$ when $S \to \infty$ cannot take the form of (3.4) even though it works as a vanilla option during the period of $[0, T_1)$. The possibility of the option being exercised becomes negligible when S is large enough.

Figure 3.6 shows the simulation paths of a risky asset with the asset price $S = 800$ at time $t = 0$. Consider a given multi-windowed barrier option A written upon this asset with strike price $K = 100$, as it can be seen in the figure, there is barely any chance that the option can be exercised.

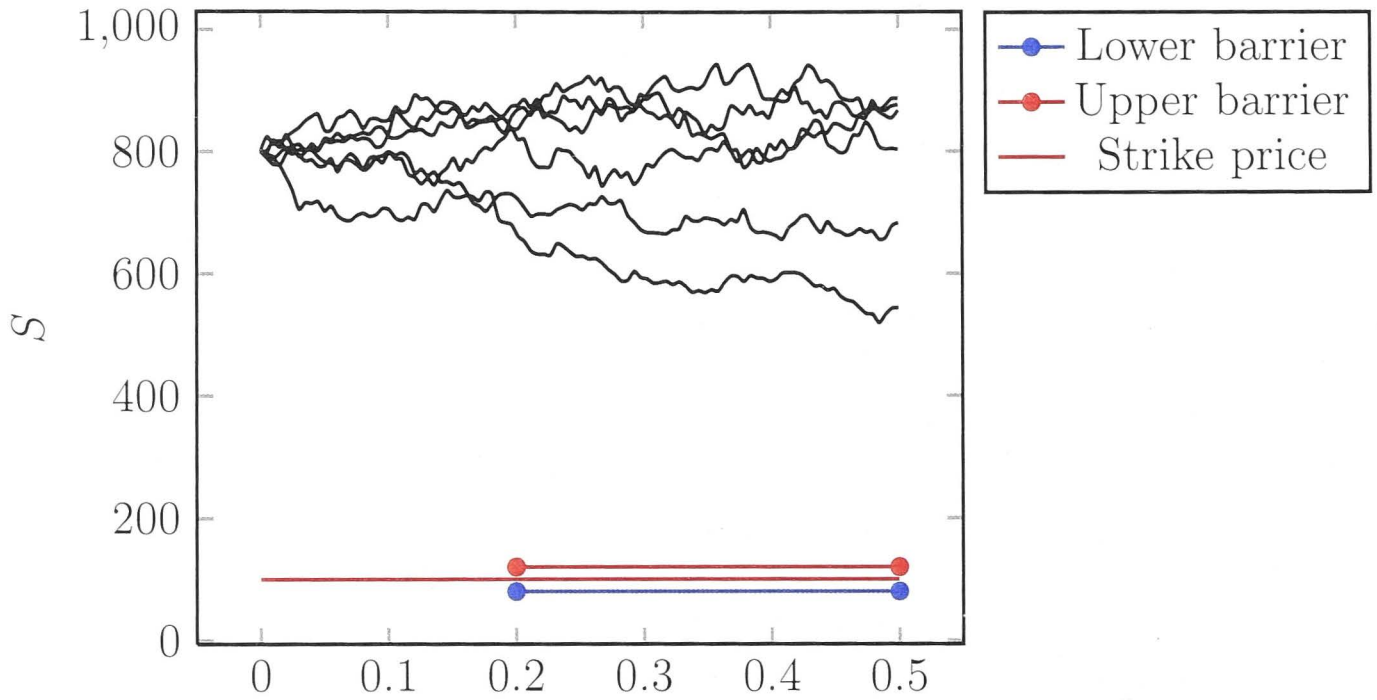

Figure 3.6: Simulation of a risky asset and a option B written on this asset. Asset price $S = 800$ at $t = 0$, the strike price $K = 100$ and the upper barrier $c_u = 120.0$, the lower barrier $c_d = 80.0$ from $t = 0.2$.

Hence, the terminal/boundary conditions for option B in $[0, T_1]$ are

$$\begin{cases} U(0,t) & = 0, \\ U(8K,t) & = 0, \\ U(S,T_1) & = U^*(S), \qquad S \in [c_d, c_u], \\ U(S,T_1) & = 0, \qquad\quad S \in (0,8K)\backslash[c_d, c_u]. \end{cases}$$

In summary, we have the boundary/initial conditions for option A:

$$\begin{aligned} &\text{when } \tau \in [0, 0.3], \quad \begin{cases} U(0,\tau) & = 0, \\ U(8K,\tau) & = 8K - Ke^{-r\tau}, \\ U(S,0) & = \max(S-K, 0), \end{cases} \\ &\text{when } \tau \in [0.3, 0.5], \quad \begin{cases} U(c_d,\tau) & = 0, \\ U(c_u,\tau) & = 0, \\ U(S,0.3) & = U^*(S), \end{cases} \end{aligned} \tag{3.5}$$

and for option B:

$$\text{when } \tau \in [0, 0.3], \quad \begin{cases} U(c_d, \tau) & = 0, \\ U(c_u, \tau) & = 0, \\ U(S, 0) & = \max(S - K, 0), \end{cases}$$

$$\text{when } \tau \in [0.3, 0.5], \quad \begin{cases} U(0, \tau) & = 0, \\ U(8K, \tau) & = 0, \\ U(S, 0.3) & = U^*(S), \quad \text{when } S \in [c_d, c_u], \\ U(S, 0.3) & = 0, \quad \text{when } S \in (0, 8K) \backslash [c_d, c_u]. \end{cases} \tag{3.6}$$

With the PDE (3.2) and the boundary/initial conditions (3.5) and (3.6), we obtain the numerical solutions of option A and option B presented in Figure 3.7 and Figure 3.8 respectively.

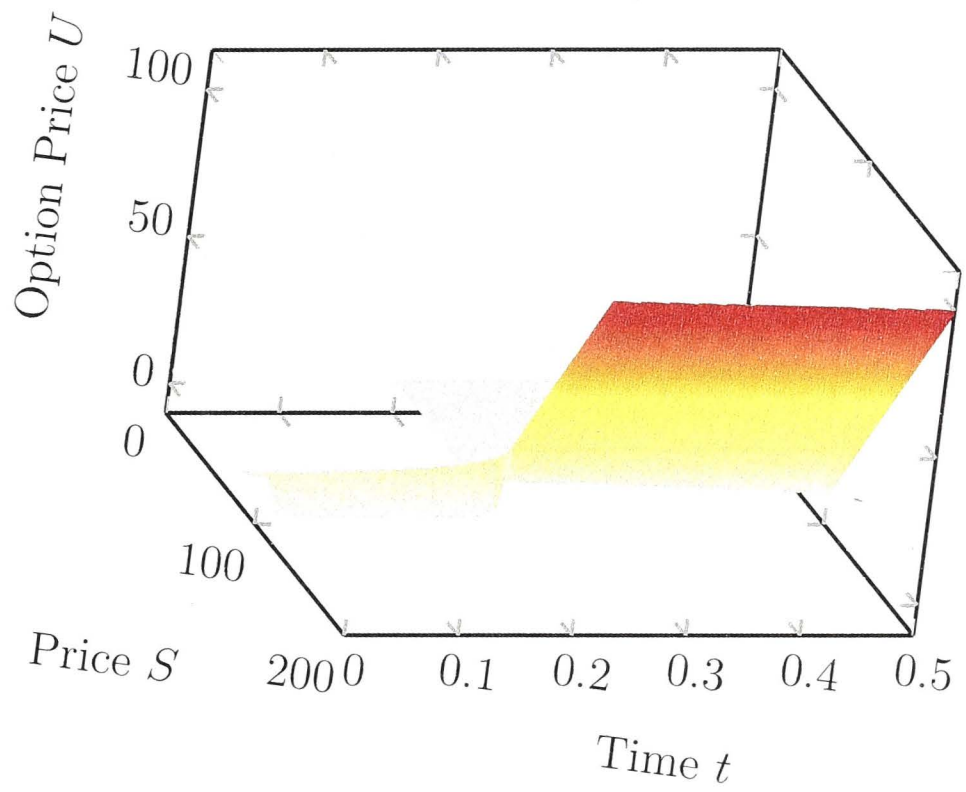

Figure 3.7: Values of multi-window barrier option A when $r = 0.10$, $\sigma = 0.2$, $T = 0.5$, $c_d = 80.0$, $c_u = 102.0$, $T_1 = 0.2$ *and* $K = 100.0$.

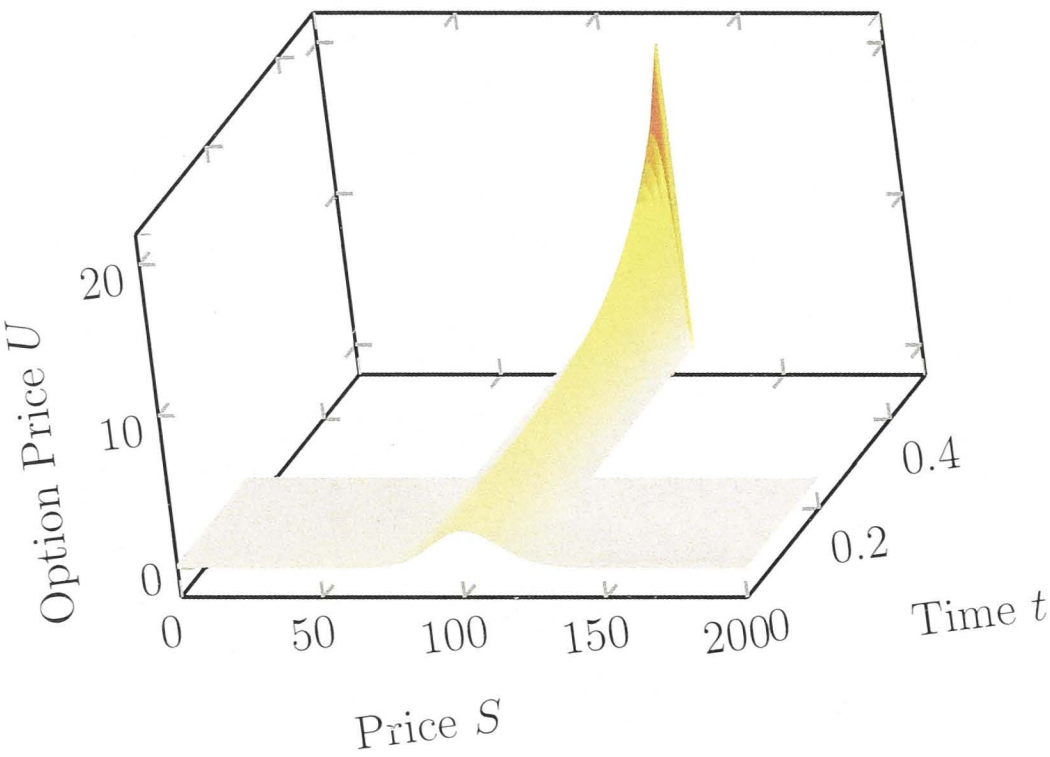

Figure 3.8: Values of multi-window barrier option B when $r = 0.10$, $\sigma = 0.2$, $T = 0.5$, $c_d = 80.0$, $c_u = 120.0$, $T_1 = 0.2$ and $K = 100.0$.

Chapter 4

FX barrier options in the Heston model

As discussed in **Chapter 1**, Black-Scholes' model has several deficiencies when it comes to matching prices in the market. A single constant volatility σ is insufficient to price options of different maturities and strikes on the same underlying asset. This is the main motivation behind using Heston's stochastic volatility model. We need a more accurate description of the volatility surface encountered in the market. For common assets other than FX, we can use the market quotes of vanilla options with different maturities and strikes to obtain the implied volatility surface. However, the quote of a FX option is not straightforward as it is always given by deltas and volatilities. This is a special convention in the FX markets.

4.1 FX market conventions and the volatility smile

4.1.1 FX market conventions

Unlike other assets, for a FX option with a spot rate S_0 and a strike rate K, there can be four quote styles in the FX markets: domestic per foreign (d/f), percentage foreign (%f), percentage domestic (%d) and foreign per domestic (f/d). They are

related with each other by the relationships:

$$U_{\%f} = \frac{U_{d/f}}{S_0},$$
$$U_{\%d} = \frac{U_{d/f}}{K},$$
$$U_{f/d} = \frac{U_{d/f}}{S_0 K}.$$

The Black-Scholes' formula of $U_{d/f}$ can be derived as follows:

Proposition 4.1 (Black-Scholes' formula for an European call/put option on foreign currency pair)**.** *For an European call/put option with strike rate K and maturity T, the price in domestic per foreign is given by*

$$U_{d/f} = \omega S_0 e^{-r_f T}\Phi(\omega d_1) - \omega K e^{-r_d T}\Phi(\omega d_2), \quad \omega = \pm 1 : call/put$$

where

$$d_1 = \frac{\ln\left(\frac{S_0}{K}\right) + \left(r_d - r_f + \frac{1}{2}\sigma^2\right)T}{\sigma\sqrt{T}},$$
$$d_2 = \frac{\ln\left(\frac{S_0}{K}\right) + \left(r_d - r_f - \frac{1}{2}\sigma^2\right)T}{\sigma\sqrt{T}},$$

and S_0 is the FX rate at the current time, r_d is the risk-free rate of the domestic currency, r_f is the risk-free rate of the foreign currency and σ is the constant volatility in Black-Scholes' model.

Proof. First let us recall that in **Chapter 2** we change the market measure $\mathbb{P}$ to a risk-neutral measure $\mathbb{Q}$ such that the discount asset price is a martingale under $\mathbb{Q}$ is a martingale. We do the same with FX rate S_t. In addition, we have to decide which currency we would like to take as the numeraire. In this case, we take the domestic currency as the numeraire. Hence under the r_d-risk-neutral measure $\mathbb{P}_d$, the FX rate S_t is given by

$$dS_t = (r_d - r_f)\, S_t dt + \sigma S_t dW_t^{\mathbb{P}_d}.$$

Then the price of a call option is

$$\begin{aligned} U_{d/f} &= e^{-r_d T} E_{\mathbb{P}_d}\left[\mathbf{1}_{\{S_T \geq K\}}(S_T - K)|S_0\right] \\ &= e^{-r_d T} E_{\mathbb{P}_d}\left[S_T \mathbf{1}_{\{S_T \geq K\}}\right] - e^{-r_d T} K \mathbb{P}_d\left(S_T \geq K\right). \end{aligned}$$

The calculation of $\mathbb{P}_d\left(S_T \geq K\right)$ is trivial since we already know the distribution of S_T under the measure $\mathbb{P}_d$. The other component requires joint distribution of S_T and $\mathbf{1}_{\{S_T \geq K\}}$, which can be solved by changing measure.

By Girsanov's Theorem (Theorem A.2), there exists a measure $\mathbb{Q}_d$ equivalent to $\mathbb{P}_d$, with the Radon-Nikodym derivative

$$\begin{aligned}\frac{d\mathbb{Q}_d}{d\mathbb{P}_d} &= \frac{S_T}{E_{\mathbb{P}_d}[S_T]} \\ &= \exp\left(\sigma W_T^{\mathbb{P}_d} - \frac{1}{2}\sigma^2 T\right),\end{aligned}$$

and the Brownian motion

$$W_t^{\mathbb{Q}_d} = W_t^{\mathbb{P}_d} + \sigma t.$$

Then we have

$$\begin{aligned}e^{-r_d T} E_{\mathbb{P}_d}\left[S_T \mathbf{1}_{\{S_T \geq K\}}\right] &= e^{-r_d T} S_0 e^{(r_d - r_f)T} E_{\mathbb{P}_d}\left[\frac{d\mathbb{Q}_d}{d\mathbb{P}_d}\mathbf{1}_{\{S_T \geq K\}}\right] \\ &= e^{-r_f T} S_0 \mathbb{Q}_d\left(S_T \geq K\right).\end{aligned}$$

Since we obtain the FX rate S_t under the measure $\mathbb{Q}_d$ given by

$$S_t = \left(r_d - r_f + \sigma^2\right) S_t dt + \sigma S_t dW_t^{\mathbb{Q}_d},$$

$\mathbb{Q}_d\left(S_T \geq K\right)$ can be calculated in the same way as $\mathbb{P}_d\left(S_T \geq K\right)$. Hence we get the formula for a call option as

$$U_{d/f} = S_0 e^{-r_f T}\Phi(d_1) - K e^{-r_d T}\Phi(d_2).$$

It is a similar approach for a put option. Then the finial result is obtained as in the proposition. □

Given a Black-Scholes' price for an option, one can calculate the change in that price for infinitesimal change in the underlying spot rate. This gives us the notation of the option delta. As there are different ways to quote the prices of FX options, there are different ways to quote the deltas:

- The *pips spot delta* is the ratio of the change in present value of the option to the change in spot rate in d/f terms:

$$\Delta_{\mathrm{S;pips}} = \lim_{\Delta S_0 \to 0} \frac{\Delta U_{d/f}}{\Delta S_0} = \omega e^{-r_f T}\Phi(\omega d_1);$$

- The *percentage spot delta* is the ratio of the change in present value of the option to the change in spot rate in $\%f$ terms:

$$\Delta_{\mathrm{S;\%}} = \lim_{\Delta S_0 \to 0} \frac{\Delta U_{\%f}}{\Delta S_0/S_0} = \omega e^{-r_d T}\frac{K}{S_0}\Phi(\omega d_2);$$

- The *pips forward delta* is the ratio of the change in future value of the option to the change in the forward rate $F_{0,T} = S_0 e^{(r_f - r_d)T}$ in d/f terms:

$$\Delta_{\text{F;pips}} = e^{r_d T} \lim_{\Delta F_{0,T} \to 0} \frac{\Delta U_{d/f}}{\Delta F_{0,T}} = \omega \Phi(\omega d_1);$$

- The *percentage forward delta* is the ratio of the change in future value of the option to the change in the forward rate in $\%f$ terms:

$$\Delta_{\text{F;\%}} = e^{r_d T} \lim_{\Delta F_{0,T} \to 0} \frac{\Delta U_{\%f}}{\Delta F_{0,T}/F_{0,T}} = \omega \frac{K}{F_{0,T}} \Phi(\omega d_2).$$

In the FX markets, one of the four delta quote types is chosen according to the currency pair. In this thesis, we consider the EUR/USD and USD/JPY currency pairs particularly. For an option on EUR/USD with maturity $T \leq 1\text{Y}$, the pips spot delta is used; for an option on USD/JPY with maturity $T \leq 1\text{Y}$, the percentage spot delta is used. Details of general rules applied on the deltas of all currency pairs can be found in [10] and [41].

In addition, the market volatility smiles of FX are not as a function of strike, but as a function of delta. Table 4.1 gives an example of how the volatilities are quoted given different maturities and deltas.

EUR/USD (spot reference 1.3456)					
Tenor	σ_{ATM} vol	$\sigma_{25-d-MS}$	$\sigma_{10-d-MS}$	$\sigma_{25-d-RR}$	$\sigma_{10-d-RR}$
1M	21.000%	0.650%	2.433%	-0.200%	-1.258%
2M	21.000%	0.750%	2.830%	-0.250%	-1.297%
3M	20.750%	0.850%	3.288%	-0.300%	-1.332%
6M	19.400%	0.900%	3.485%	-0.500%	-1.408%
1Y	18.250%	0.950%	3.806%	-0.600%	-1.359%
2Y	17.677%	0.850%	3.208%	-0.562%	-1.208%

Table 4.1: EUR/USD volatility market data, 15th Dec, 2008

The data in Table 4.1 describe three aspects of the volatility smile: at-the-money (ATM), market strangle (MS) and risk reversal (RR). We can derive the full volatility smile $\sigma_X(K)$ from them. In the FX markets, the concept of at-the-money is that we buy a straddle (long position of both a call and a put) with the same strike K_{ATM} so that the net delta is 0.

Let Δ_Q be the notation of whichever chosen in $\{\Delta_{\text{S;pips}}, \Delta_{\text{S;\%}}, \Delta_{\text{F;pips}}, \Delta_{\text{F;\%}}\}$, and $\Delta_Q(\omega, K, T, \sigma)$ be the delta of a call/put option ($\omega = \pm 1$ respectively) with different strike K, maturity T and volatility σ. Similarly, let $U(\omega, K, T, \sigma) = U_{d/f}$. Then the strike K_{ATM} is chosen such that

$$\Delta_Q(+1, K_{\text{ATM}}, T, \sigma_{\text{ATM}}) + \Delta_Q(-1, K_{\text{ATM}}, T, \sigma_{\text{ATM}}) = 0.$$

After solving the equation, we obtain the strike K_{ATM} in d/f and $\%f$ terms respectively:

$$K_{\text{ATM};d/f} = F_{0,T} \exp\left(\frac{1}{2}\sigma^2_{\text{ATM}} T\right),$$

$$K_{\text{ATM};\%f} = F_{0,T} \exp\left(-\frac{1}{2}\sigma^2_{\text{ATM}} T\right).$$

The concept of the market strangle, e.g., $\sigma_{25-d-MS}$, is that we buy a call and a put with different strikes $K_{25-d-C-MS}$ and $K_{25-d-P-MS}$ respectively such that

$$\begin{aligned} &U\left(+1, K_{25-d-C-MS}, T, \sigma_{\text{ATM}} + \sigma_{25-d-MS}\right) \\ &\quad + U(-1, K_{25-d-P-MS}, T, \sigma_{\text{ATM}} + \sigma_{25-d-MS}) = \\ &U\left(+1, K_{25-d-C-MS}, T, \sigma_X(K_{25-d-C-MS})\right) \\ &\quad + U\left(-1, K_{25-d-P-MS}, T, \sigma_X(K_{25-d-P-MS})\right), \end{aligned} \tag{4.1}$$

and

$$\begin{aligned} \Delta_Q(+1, K_{25-d-C-MS}, T, \sigma_{\text{ATM}} + \sigma_{25-d-MS}) &= +0.25, \\ \Delta_Q(-1, K_{25-d-P-MS}, T, \sigma_{\text{ATM}} + \sigma_{25-d-MS}) &= -0.25. \end{aligned}$$

The risk reversal of 25-delta, i.e., $\sigma_{25-d-RR}$ is given by

$$\sigma_{25-d-RR} = \sigma_X(K_{25-d-C}) - \sigma_X(K_{25-d-P}), \tag{4.2}$$

where K_{25-d-C} and K_{25-d-P} are strikes for a call and a put, and satisfy

$$\begin{aligned} \Delta_Q(+1, K_{25-d-C}, T, \sigma_X(K_{25-d-C})) &= +0.25, \\ \Delta_Q(-1, K_{25-d-P}, T, \sigma_X(K_{25-d-P})) &= -0.25. \end{aligned} \tag{4.3}$$

25-delta call and 25-delta put are the most liquid in the FX markets. They are import benchmark strikes for constructing FX volatility smiles. 10-delta call and 10-delta put are less liquid but still available as shown in Table 4.1.

4.1.2 Interpolation of the volatility smile

In this section, we demonstrate deriving the volatility smiles for the currency pairs using the volatilities of 25-delta. There are several approaches to do this, and we choose to construct a polynomial of $\sigma_X(K)$ to interpolate volatilities over the strike K and then calibrate the coefficients in the polynomial with the market data. The polynomial is given by

$$\sigma_X(K) = \exp\left[f(\ln(F_{0,T})/K)\right] \tag{4.4}$$

with

$$f(x) = c_0 + c_1\delta(x) + c_2\delta(x)^2, \tag{4.5}$$

where $\delta_0 = \exp(c_0)$ and

$$\delta(x) = \Phi\left(x/\delta_0\sqrt{T}\right).$$

From last section we know that $\sigma_X(K)$ must satisfy (4.2) and the condition

$$\sigma_X(K_{\text{ATM}}) = \sigma_{\text{ATM}}. \tag{4.6}$$

In addition, we introduce the 25-delta smile strangle given by

$$\sigma_{25-d-SS} = \frac{1}{2}\left[\sigma_X(K_{25-d-C}) + \sigma_X(K_{25-d-P})\right] - \sigma_X(K_{\text{ATM}}) \tag{4.7}$$

where K_{25-d-C} and K_{25-d-P} are the solutions of (4.3).

From (4.2), (4.6) and (4.7), we obtain

$$\begin{aligned}
\sigma_X(K_{ATM}) &= \sigma_{ATM}, \\
\sigma_X(K_{25-d-C}) &= \frac{1}{2}\left[2(\sigma_{25-d-SS} + \sigma_{ATM}) + \sigma_{25-d-RR}\right], \\
\sigma_X(K_{25-d-P}) &= \frac{1}{2}\left[2(\sigma_{25-d-SS} + \sigma_{ATM}) - \sigma_{25-d-RR}\right].
\end{aligned} \tag{4.8}$$

Then the coefficients in (4.5) and $\sigma_{25-d-SS}$ are numerically chosen by the Levenberg-Marquardt algorithm such that the values of $\sigma_X(K)$ obtained with (4.4) satisfy the conditions in (4.1) and (4.8).

With the market volatilities of EUR/USD in Table 4.1, we obtain the coefficients of the polynomial (4.5): $c_0 = -1.46041$, $c_1 = -1.085455$ and $c_2 = 1.222756$. Then we back out the smile strikes and volatilities as shown in Table 4.2 and Figure 4.1.

We can see that the $\sigma_X(K_{\text{ATM}})$ is correct compared with the quote in Table 4.1 and $\sigma_{25-d-RR} = \sigma_{25-d-C} - \sigma_{25-d-P} = -0.60\%$. Also the market volatilities match the derived volatility smile.

K_{25-d-P}	1.2034	19.50%
$K_{25-d-P-MS}$	1.2050	19.48%
K_{ATM}	1.3620	18.25%
K_{25-d-C}	1.5410	18.90%
$K_{25-d-C-MS}$	1.5449	18.92%

Table 4.2: 1Y EUR/USD Volatility Smile

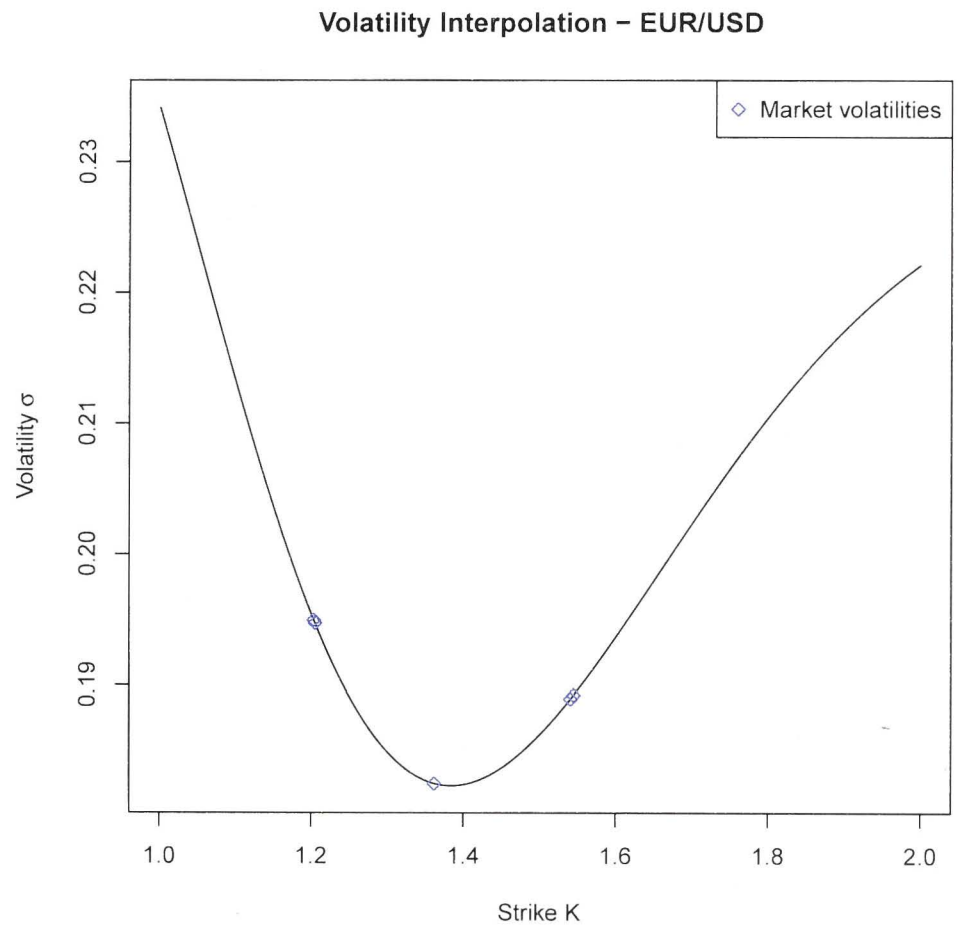

Figure 4.1: 1Y EUR/USD Volatility Curve

4.2 FX vanilla options

4.2.1 Closed-form formula

In Heston's stochastic volatility model, the volatility of the asset price is another stochastic process. The SDE of the FX exchange rate S under r_d-risk-neutral measure $\mathbb{P}_d$ is given by:

$$\begin{aligned} dS_t &= (r_d - r_f)S_t dt + \sqrt{v_t} S_t dZ_t^{\mathbb{P}_d}, \\ dv_t &= \kappa(\theta - v_t)dt + \xi\sqrt{v_t} dW_t^{\mathbb{P}_d} \end{aligned} \tag{4.9}$$

with the correlation ρ between the Brownian motions $Z_t^{\mathbb{P}_d}$ and $W_t^{\mathbb{P}_d}$ defined by

$$\rho dt = d\left\langle Z^{\mathbb{P}_d}, W^{\mathbb{P}_d}\right\rangle_t, \quad \rho \in [-1, 1].$$

From the proof of Proposition 4.1 we know that the present value of an European call option with strike K and maturity T follows

$$U_{d/f} = S_0 e^{-r_f}\mathbb{Q}_d(S_T \geq K) - Ke^{-r_d}\mathbb{P}_d(S_T \geq K), \tag{4.10}$$

where $\mathbb{Q}_d$ is an equivalent martingale measure of $\mathbb{P}_d$. Both cumulative probabilities can be found by inverting the characteristic function $\phi_j(x, v, t; \varepsilon)$, $j = 1, 2$:

$$\mathbb{Q}_d(x \geq 0) = \frac{1}{2} + \frac{1}{\pi}\int_0^\infty \Re\left[\frac{\phi_1(x, v, 0; \varepsilon)}{i\varepsilon}\right] d\varepsilon,$$

$$\mathbb{P}_d(x \geq 0) = \frac{1}{2} + \frac{1}{\pi}\int_0^\infty \Re\left[\frac{\phi_2(x, v, 0; \varepsilon)}{i\varepsilon}\right] d\varepsilon,$$

where

$$x = \ln\left(\frac{S_T}{K}\right).$$

The characteristic functions can not be obtained immediately though, we can use the PDE approach in [23]. We rewrite (4.9) as

$$d\begin{pmatrix} S_t \\ v_t \end{pmatrix} = \begin{pmatrix} (r_d - r_f)S_t \\ \kappa(\theta - v_t) \end{pmatrix} dt + \begin{pmatrix} \sqrt{v_t}S_t & 0 \\ \xi\rho\sqrt{v_t} & \xi\sqrt{v_t(1-\rho^2)} \end{pmatrix} d\begin{pmatrix} X_1^{\mathbb{P}_d} \\ X_2^{\mathbb{P}_d} \end{pmatrix},$$

where $X_1^{\mathbb{P}_d}$ and $X_2^{\mathbb{P}_d}$ are two independent Brownian motions. By Feynman-Kac's Formula (Theorem A.7), the PDE of the call option price $U = U(S, v, t)$ is

$$\frac{\partial U}{\partial t} + \left[(r_d - r_f)S\frac{\partial U}{\partial S} + \kappa(\theta - v)\frac{\partial U}{\partial v}\right] + \left[\frac{1}{2}vS^2\frac{\partial^2 U}{\partial S^2} + \xi\rho vS\frac{\partial^2 U}{\partial S\partial v} + \frac{1}{2}\xi^2 v\frac{\partial^2 U}{\partial v^2}\right] - r_d U = 0 \tag{4.11}$$

with the terminal condition

$$U(S, v, T) = \max(S - K, 0),$$

and (4.10) is the solution.

Let $P_1 = \mathbb{Q}_d(x \geq 0)$ and $P_2 = \mathbb{P}_d(x \geq 0)$. After substituting (4.10) into (4.11), we have P_1 and P_2 satisfying the PDEs

$$\frac{\partial P_j}{\partial t} + \left[(r_d - r_f + a_j v)\frac{\partial P_j}{\partial x} + (\kappa\theta - b_j v)\frac{\partial P_j}{\partial v}\right] + \left[\frac{1}{2}v\frac{\partial^2 P_j}{\partial x^2} + \xi\rho v\frac{\partial^2 P_j}{\partial x\partial v} + \frac{1}{2}\xi^2 v\frac{\partial^2 P_j}{\partial v^2}\right] = 0, \tag{4.12}$$

for $j = 1, 2$, where

$$a_1 = \frac{1}{2},\ a_2 = -\frac{1}{2},\ b_1 = \kappa - \rho\xi,\ b_2 = \kappa.$$

The PDEs (4.12) subject to the terminal condition

$$P_j = \mathbf{1}_{\{x \geq 0\}}, \quad j = 1, 2.$$

Hence the characteristic functions satisfy the Fokker-Planck forward equation:

$$\frac{\partial \phi_j}{\partial t} + \left[(r_d - r_f + a_j v)\frac{\partial \phi_j}{\partial x} + (\kappa\theta - b_j v)\frac{\partial \phi_j}{\partial v}\right] + \left[\frac{1}{2}v\frac{\partial^2 \phi_j}{\partial x^2} + \xi\rho v\frac{\partial^2 \phi_j}{\partial x \partial v} + \frac{1}{2}\xi^2 v\frac{\partial^2 \phi_j}{\partial v^2}\right] = 0$$

with the terminal condition

$$\phi_j(x, v, T) = e^{ix\varepsilon}.$$

The solutions of the characteristic functions are

$$\phi_j(x, v, t; \varepsilon) = \exp\left[C(T - t; \varepsilon) + D(T - t; \varepsilon) + ix\varepsilon\right],$$

where

$$C(\tau; \varepsilon) = \frac{\kappa\theta}{\xi^2}\left[(b_j - \rho\xi\varepsilon i) - 2\ln\left(\frac{1 - ge^{d\tau}}{1 - g}\right)\right],$$
$$D(\tau; \varepsilon) = \frac{b_j - \rho\xi\varepsilon i + d}{\xi^2}\left(\frac{1 - e^{d\tau}}{1 - ge^{d\tau}}\right),$$

with $\tau = T - t$, and

$$g = \frac{b_j - \rho\xi\varepsilon i + d}{b_j - \rho\xi\varepsilon i - d},$$
$$d = \sqrt{(\rho\xi\varepsilon i - b_j)^2 - \xi^2(2a_j\varepsilon i - \varepsilon^2)}.$$

We can then extract the implied volatility σ^{implied} from the option price $U_{d/f}$. In last section, we derive the volatility smile σ^{market} according to the market volatilities based on deltas. Figure 4.1 gives a series of strike rates and volatilities. Accordingly, we can obtain the values of parameters in (4.9) by minimizing the root mean square error between σ^{implied} and σ^{market}

$$\min_{\kappa,\theta,\xi,\rho,v_0} \sqrt{\frac{1}{N}\sum_{j=1}^{N}\left(\sigma_j^{\text{implied}} - \sigma_j^{\text{market}}\right)^2}.$$

There are several approaches to achieve accuracy and to spend less time on calibrating the parameters. The details can be found in [25] and [38]. The calibration is a quite interesting topic of research and it is beyond the scope of this thesis.

We use the parameters in Table 4.3 to price an European call option on EUR/JPY.

Parameter	r_d(JPY)	r_f(EUR)	ξ	ρ	κ	θ	K	T
Value	0.0117	0.0346	0.5	-0.1	2.5	0.06	100.0	0.5

Table 4.3: EUR/JPY vanilla call parameter values

4.2.2 Numerical solution using the PDE approach

We change the PDE (4.11) into the form

$$\frac{\partial U}{\partial \tau} - \left[(r_d - r_f)S\frac{\partial U}{\partial S} + \kappa(\theta - v)\frac{\partial U}{\partial v}\right] - \left[\frac{1}{2}vS^2\frac{\partial^2 U}{\partial S^2} + \xi\rho vS\frac{\partial^2 U}{\partial S \partial v} + \frac{1}{2}\xi^2 v\frac{\partial^2 U}{\partial v^2}\right] + r_d U = 0, \quad (4.13)$$

where $\tau = T - t$, with the initial condition

$$U(S, v, 0) = \max(S - K, 0).$$

To obtain the solution of (4.13), we must specify appropriate boundary conditions. There are different ways to do this. In [18], the boundary conditions for a general call option are given by

$$\begin{cases} \dfrac{\partial U}{\partial \tau} - \kappa(\theta - v)\dfrac{\partial U}{\partial v} - \dfrac{1}{2}\xi^2 v\dfrac{\partial^2 U}{\partial v^2} + r_d U = 0, & S = 0 \\ U = Se^{-r_f \tau}, & S \to \infty \\ \dfrac{\partial U}{\partial \tau} - (r_d - r_f)S\dfrac{\partial U}{\partial S} - \kappa\theta\dfrac{\partial U}{\partial v} + r_d U = 0, & v = 0 \\ \dfrac{\partial U}{\partial \tau} - (r_d - r_f)S\dfrac{\partial U}{\partial S} - \dfrac{1}{2}vS^2\dfrac{\partial^2 U}{\partial S^2} + r_d U = 0, & v \to \infty. \end{cases}$$

They are determined by setting the variables in (4.11) to be the boundary values. Its aim is to apply this PDE formulation to a wide variety of two factors options. Once the type of the options is chosen, the boundary conditions can be solved

numerically except that the condition of $S \to \infty$ is treated as a Dirichlet boundary condition.

If we consider a most important property of the vanilla options: the option has no value if the price of underlying asset is 0, then we have $U = 0$ as the boundary condition when $S = 0$. This gives the analytic solution of the boundary condition

$$\frac{\partial U}{\partial \tau} - (r_d - r_f)S\frac{\partial U}{\partial S} - \frac{1}{2}vS^2\frac{\partial^2 U}{\partial S^2} + r_d U = 0, \quad v \to \infty, \tag{4.14}$$

which is one of the boundary conditions used in [23].

Now we show how to obtain this boundary condition, when $v \to \infty$, the only condition which makes (4.14) valid on the domain where $S \neq 0$ is

$$\frac{\partial^2 U}{\partial S^2} = 0.$$

Accordingly, (4.14) is simplified as

$$\frac{\partial U}{\partial \tau} - (r_d - r_f)S\frac{\partial U}{\partial S} + r_d U = 0, \quad v \to \infty$$

and U has a solution of the form

$$U = a(\tau)S + b(\tau). \tag{4.15}$$

Next we substitute (4.15) into (4.14) to obtain

$$\left[\frac{\partial a(\tau)}{\partial \tau} + r_f a(\tau)\right] S + \frac{\partial b(\tau)}{\partial \tau} + r_d b(\tau) = 0.$$

Hence we have two ODEs

$$\begin{aligned} \frac{\partial a(\tau)}{\partial \tau} + r_f a(\tau) &= 0, \\ \frac{\partial b(\tau)}{\partial \tau} + r_d b(\tau) &= 0. \end{aligned} \tag{4.16}$$

Since U has to satisfy the boundary conditions

$$\begin{cases} U = 0, & S = 0, \\ U = Se^{-r_f \tau}, & S \to \infty, \end{cases}$$

we have the solutions for (4.16)

$$\begin{aligned} a(\tau) &= e^{-r_f \tau}, \\ b(\tau) &= 0. \end{aligned}$$

Thus the analytic solution for (4.14) is

$$U = Se^{-r_f\tau}.$$

Hence the boundary conditions for the vanilla call option are given as

$$\begin{cases} U & = 0, & S = 0, \\ \dfrac{\partial U}{\partial S} & = e^{-r_f\tau}, & S \to \infty, \\ \dfrac{\partial U}{\partial \tau} & - (r_d - r_f)S\dfrac{\partial U}{\partial S} - \kappa\theta\dfrac{\partial U}{\partial v} + r_d U = 0, & v = 0, \\ U & = Se^{-r_f\tau}, & v \to \infty. \end{cases} \tag{4.17}$$

Now we consider the other property of the vanilla option: if $v = 0$, the option value is independent of the volatility, i.e., $\frac{\partial U}{\partial v} = 0$. Then the boundary condition when $v = 0$ in (4.17) is simplified as

$$\frac{\partial U}{\partial \tau} - (r_d - r_f)S\frac{\partial U}{\partial S} + r_d U = 0, \quad v = 0. \tag{4.18}$$

By employing the similar procedure above, we obtain the analytic solution of (4.18) as

$$U = \max(Se^{-r_f\tau} - Ke^{-r_d\tau}, 0), \quad v = 0.$$

This allows us to obtain the boundary conditions given in [52] as

$$\begin{cases} U & = 0, & S = 0, \\ \dfrac{\partial U}{\partial S} & = e^{-r_f\tau}, & S \to \infty, \\ U & = \max(Se^{-r_f\tau} - Ke^{-r_d\tau}, 0), & v = 0, \\ U & = Se^{-r_f\tau}, & v \to \infty. \end{cases}$$

We choose not to use the analytic solutions for the boundary conditions, because:

i it is not that easy to obtain such solutions for some barrier options, especially multi-windowed barrier options;

ii it is very difficult if not impossible to obtain them when we price a option in the multi-dimensional Heston model, which we will see in the next chapter.

Hence, we use the boundary conditions:

$$\begin{cases} U & = 0, & S = 0, \\ \dfrac{\partial U}{\partial S} & = e^{-r_f \tau}, & S \to \infty, \\ \dfrac{\partial U}{\partial v} & = 0, & v = 0, \\ \dfrac{\partial U}{\partial v} & = 0, & v \to \infty. \end{cases}$$

Performing the change of variable $x = \ln S$ and setting $U = U(x, v, \tau)$, the PDE (4.13) can be written as

$$\frac{\partial U}{\partial \tau} - \left[(r_d - r_f - \frac{1}{2}v)\frac{\partial U}{\partial x} + \kappa(\theta - v)\frac{\partial U}{\partial v}\right] - \left[\frac{1}{2}v\frac{\partial^2 U}{\partial x^2} + \xi\rho v\frac{\partial^2 U}{\partial x \partial v} + \frac{1}{2}\xi^2 v\frac{\partial^2 U}{\partial v^2}\right] + r_d U = 0.$$

This PDE can be transformed into the divergence form,

$$\frac{\partial U}{\partial t} - \mathbf{V} \cdot \nabla U - \nabla \cdot \mathbf{M} \nabla U + r_d U = 0,$$

where

$$\nabla = \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial v} \end{pmatrix},$$

$$\mathbf{V} = \begin{pmatrix} r_d - r_f - \frac{1}{2}v - \frac{1}{2}\xi\rho \\ \kappa(\theta - v) - \frac{1}{2}\xi^2 \end{pmatrix},$$

and

$$\mathbf{M} = \frac{1}{2}v \begin{pmatrix} 1 & \xi\rho \\ \xi\rho & \xi^2 \end{pmatrix}.$$

We choose the space domain of $x \times v$ as $[-5, 7] \times [0, 5]$ so that the initial/boundary conditions for the vanilla call option are

$$\begin{cases} U & = \max(e^x - K, 0), & \tau = 0, \\ U & = 0, & x = -5, \\ \dfrac{\partial U}{\partial x} & = e^{x - r_f \tau}, & x = 7, \\ \dfrac{\partial U}{\partial v} & = 0, & v = 0 \text{ or } v = 5. \end{cases}$$

We discretise the space domain with a non-uniform mesh which has mesh increment $\Delta x = 0.01$ when $x \in [-5, 5]$, $\Delta v = 0.01$ when $v \in [0, 1]$ and $\Delta x = \Delta v = 0.1$ otherwise. Let the time step be $\Delta\tau = 0.025$, we obtain the numerical results as shown in Figure 4.2.

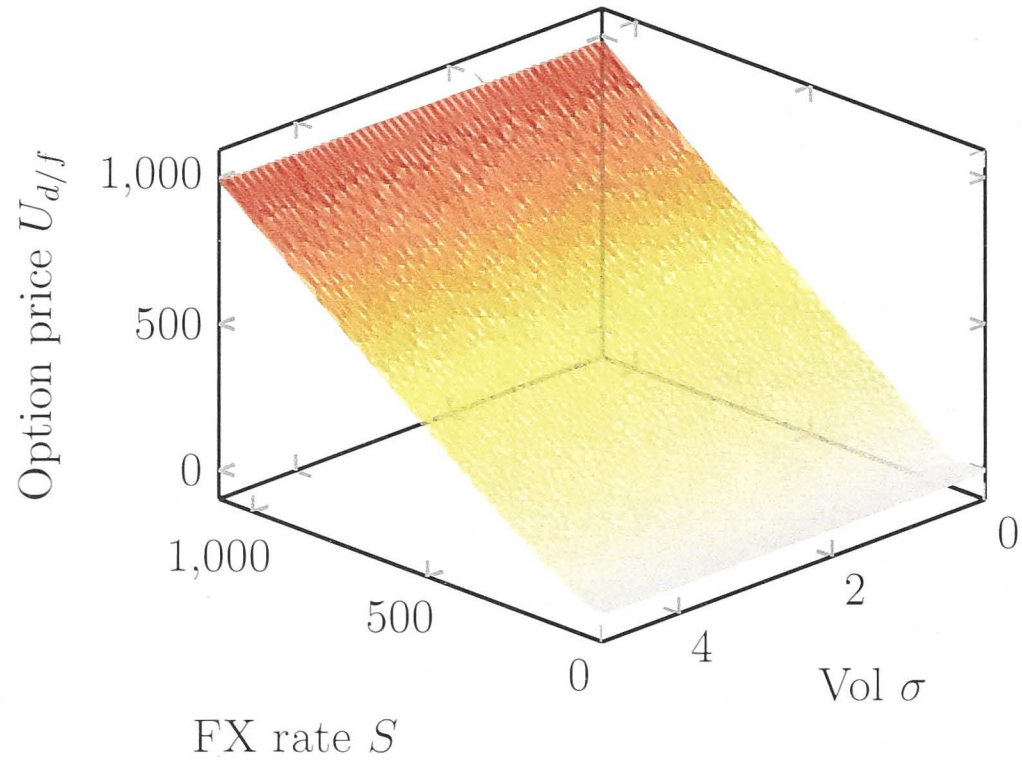

Figure 4.2: Numerical results of EUR/JPY vanilla call option. The mesh is non-uniform with $\Delta x = 0.01$ *when* $x \in [-5, 5]$, $\Delta v = 0.01$ *when* $v \in [0, 1]$ *and* $\Delta x = \Delta v = 0.1$ *otherwise. The time step is* $\Delta\tau = 0.025$.

The analytic results [1] computed on the same mesh are shown in Figure 4.3. Note that we do not use the formulae originally derived by Heston but the ones as in [2], since the later are more stable when it comes to the situation where Feller's condition is violated.

The difference between the numerical results and the analytic results on the whole space domain is shown in Figure 4.4. We can see that we do not achieve good accuracy over the whole domain in the sense of the sup norm given by

$$\|U_{\text{numerical}} - U_{\text{analytic}}\|_\infty = 7.6107.$$

The difference is big near the boundary at $v = 5$ and $x = 7$ (i.e., $S = 1096.63$).

However, in the option pricing, we only care the results around the strike price where $S \in [K, 2K]$ and $v \in [0, 1]$. The difference on this domain is shown in Figure 4.5 and we achieve the accuracy in percentage sup norm given by

$$\left\| \frac{U_{\text{numerical}} - U_{\text{analytic}}}{U_{\text{analytic}}} \right\|_\infty = 3.21 \times 10^{-3}.$$

[1] R code of pricing vanilla option with the semi-closed formula in the Heston model is from Dr. Dale Roberts.

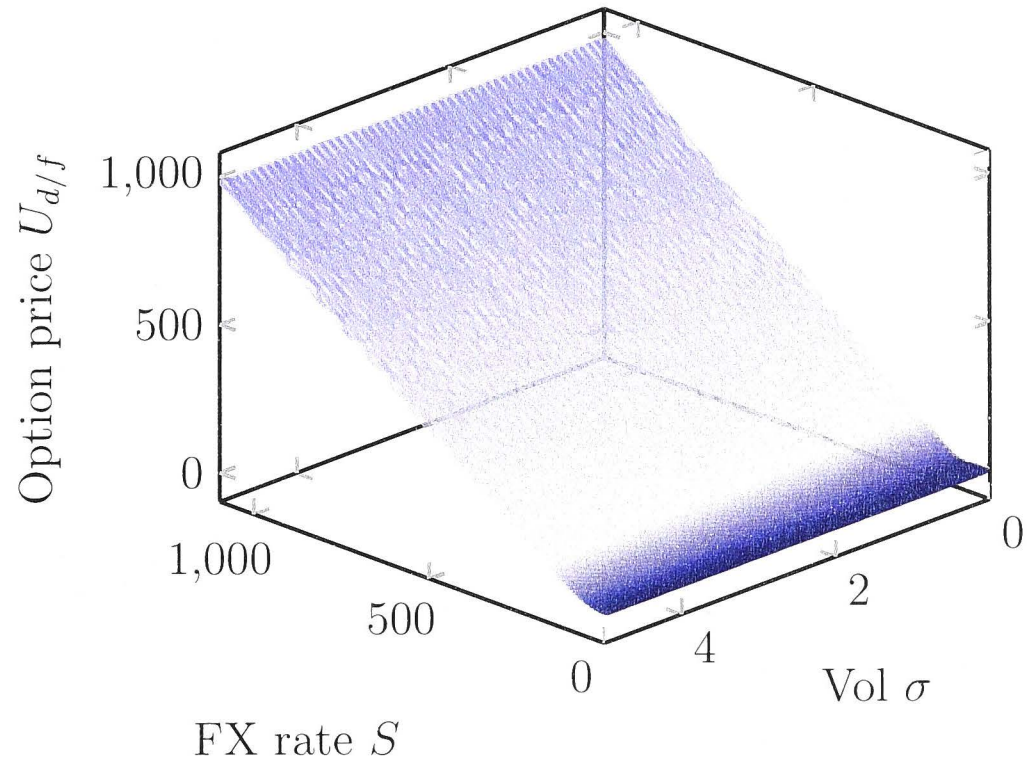

Figure 4.3: Analytic results of EUR/JPY vanilla call option. The mesh is non-uniform with $\Delta x = 0.01$ when $x \in [-5, 5]$, $\Delta v = 0.01$ when $v \in [0, 1]$ and $\Delta x = \Delta v = 0.1$ otherwise.

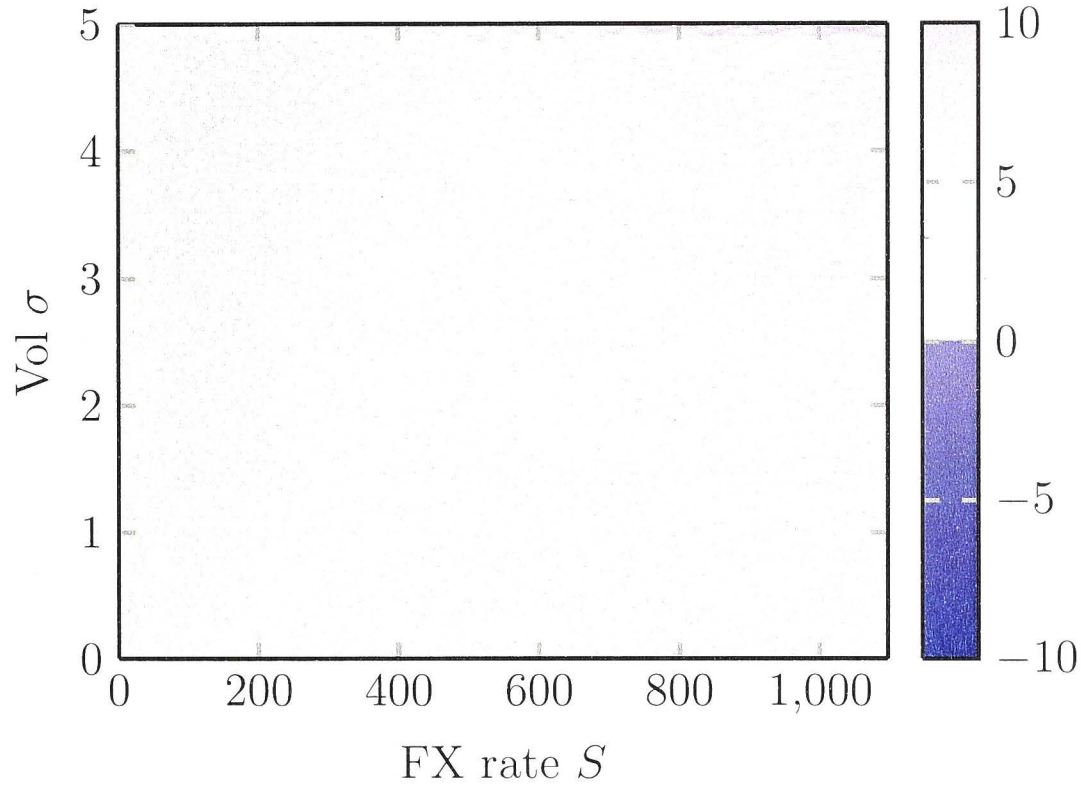

Figure 4.4: Difference between the numerical results and the analytic results. The color indicates the value of $(U_{numerical} - U_{analytic})$.

4.3 FX multi-windowed barrier option

Now we consider a multi-windowed barrier option on EUR/JPY as indicated in Figure 3.5 in **Chapter 3**, i.e., option B. We employ the same approach as in Black-Scholes' model. Let $T_1 = 0.3$, $c_d = 90.0$ and $c_u = 180.0$. We have the

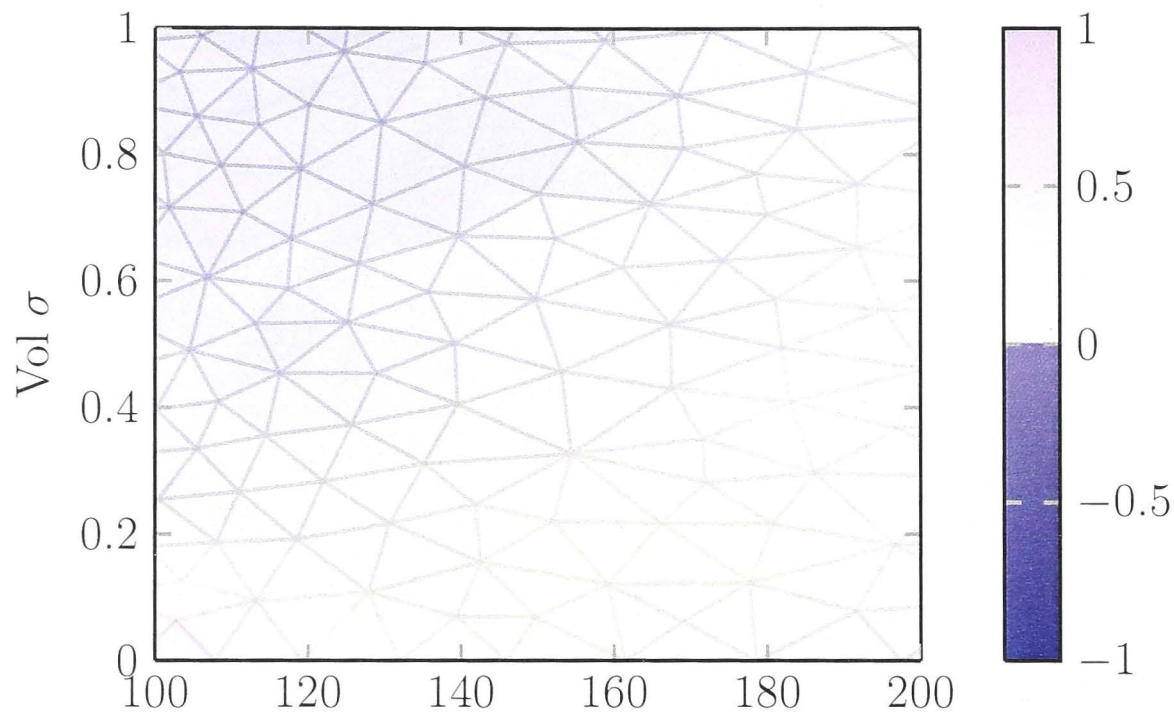

Figure 4.5: Difference between the numerical results and the analytic results on $[K, 2K] \times [0, 1]$. *The color indicates the value of* $(U_{numerical} - U_{analytic})$.

initial/boundary conditions for such an option:

$$\text{when } \tau \in [0, 0.2], \quad \begin{cases} U & = \max(e^x - K, 0), \quad \tau = 0, \\ U & = 0, \qquad\qquad\quad x = \ln c_d \text{ or } x = \ln c_u, \\ \dfrac{\partial U}{\partial v} & = 0, \qquad\qquad\quad v = 0 \text{ or } v = 5, \end{cases}$$

$$\text{when } \tau \in [0.2, 0.5], \quad \begin{cases} U & = U^*(x), \quad \tau = 0.2, \quad x \in [\ln c_d, \ln c_u], \\ U & = 0, \qquad \tau = 0.2, \quad x \in (-5, 7) \backslash [\ln c_d, \ln c_u], \\ U & = 0, \qquad x = -5 \text{ or } x = 7, \\ \dfrac{\partial U}{\partial v} & = 0, \qquad v = 0 \text{ or } v = 5, \end{cases}$$

where $U^*(x)$ is the solution of U at time $T_1 = 0.3$.

When $t \in [0, T_1]$, the space domain $x \times v$ is defined by $[\ln c_d, \ln c_u] \times [0, 5]$. We discretise this domain with a uniform mesh where $\Delta x = \Delta v = 0.01$. Let the time step be $\Delta\tau = 0.005$. Then we obtain the solution $U^*(x)$ at T_1.

When $t \in [T_1, 0.5]$, the space domain is $[-5, 7] \times [0, 5]$. We use the non-uniform mesh where $\Delta x = \Delta v = 0.01$ when $x \in [-5, \ln c_u]$ and $v \in [0, 1]$, and $\Delta x = \Delta v = 0.1$ otherwise. The solution $U^*(x)$ is interpolated on this space domain as the initial condition. Let the time step be $\Delta\tau = 0.005$. We obtain the final solution of $U_{d/f}$.

We choose the volatility to be $v = 0.2$ and plot the solution $U_{d/f}$ as a function of time to maturity t and the exchange rate of EUR/JPY in Figure 4.6.

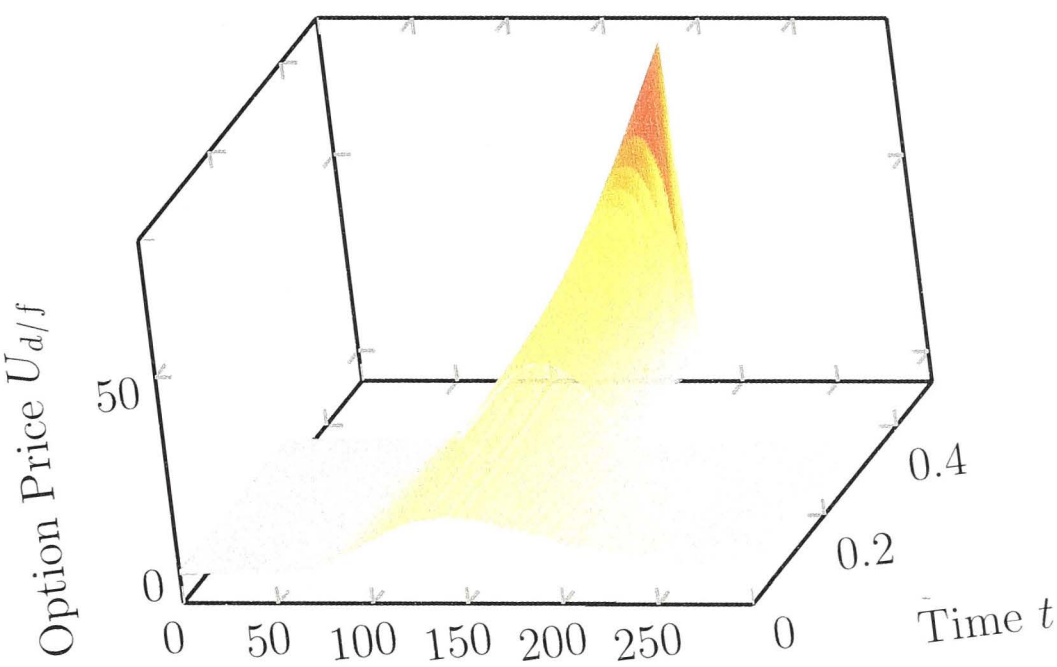

Figure 4.6: Value of multi-window option B when $c_d = 90.0$, $c_u = 180.0$ *and* $T_1 = 0.3$. *The mesh is uniform with* $\Delta x = \Delta v = 0.01$ *when* $t \in [0.3, 0.5]$; *the mesh is non-uniform with* $\Delta x = \Delta v = 0.01$ *where* $x \in [-5, \ln c_u]$ *and* $v \in [0, 1]$, *and* $\Delta x = \Delta v = 0.1$ *otherwise when* $t \in [0, 0.3]$. *The time step is* $\Delta\tau = 0.005$.

Chapter 5

FX barrier options in the multi-dimensional Heston model

The steps of pricing a multi-windowed FX barrier option are demonstrated in **Chapter 4**:

i Interpolate the market volatilities and obtain the market volatility smile;

ii Calibrate the parameters in (4.9) using the market volatility smile;

iii Generate the PDE and determine the boundary/initial conditions according to the barrier option type;

iv Obtain the numerical solutions of the PDE using the finite element method.

In addition, we also know that FX options are very different from other options in the market in the sense that the option price is not represented by 'money'. In this chapter, we discuss another fact special in the FX market, which is so-called triangle relationship between the multiple currency pairs. This gives us the context to price FX options in the multi-dimensional Heston model.

5.1 Calibration of multi-currency pairs

Let us consider three currency pairs EUR/USD, USD/JPY and EUR/JPY liquidly traded in the market. The triangle relationship between these three currency pairs is defined by

$$S_t^{\text{EUR/JPY}} = S_t^{\text{USD/JPY}} S_t^{\text{EUR/USD}},$$

and it is shown in Figure 5.1. This relationship guarantees that there is no arbitrage opportunity in the market.

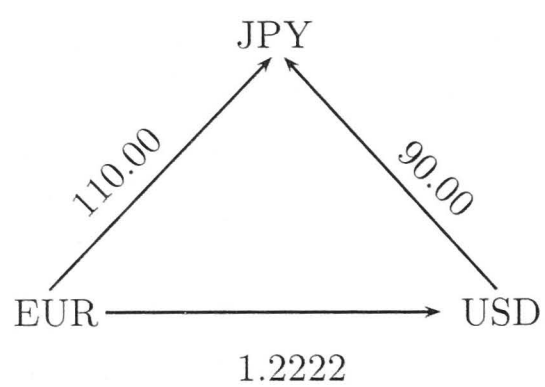

Figure 5.1: Currency triangle relationship

The triangle relationship is defined by the correlation between the two main FX rates EUR/USD and USD/JPY as

$$\rho_{\text{EUR/USD-USD/JPY}} = \frac{\sigma^2_{\text{EUR/JPY}} - \sigma^2_{\text{EUR/USD}} - \sigma^2_{\text{USD/JPY}}}{2\sigma_{\text{EUR/USD}}\sigma_{\text{USD/JPY}}}.$$

It involves the volatility smile of the third currency pair EUR/JPY, which means that any option written on the two currency pairs is sensitive to the third one.

The correlation $\rho_{\text{EUR/USD-USD/JPY}}$ can not be constant for the existence of the volatility skew. One approach to determine $\rho_{\text{EUR/USD-USD/JPY}}$ is to make it a polynomial with respect to time t and strike K so that we can calibrate it according to the volatility smiles of three currency pairs. Besides, we might calibrate the unknown coefficient respectively for each currency pair. Then finally we are able to price this option.

There are two deficiencies with this approach. Firstly, it is a "case by case" approach. Before the calibration, we have to decide which currency will be deemed as the base currency, because this pricing model can not treat all the currencies symmetrically. Secondly, this calibration process is both time and computation consuming. Each time when we have an option written on two or more different currency pairs, we have to repeat the same procedure. Moreover, if an option is written on two currency pairs without a common base currency, e.g. , EUR/USD and AUD/JPY, then we have a more complicated tetrahedron relationship as shown in Figure 5.2. This makes the calibration more difficult.

Hence, it is highly preferred to have a good pricing model, which can a) reproduce all the vanilla option markets observed in the real world; b) treat

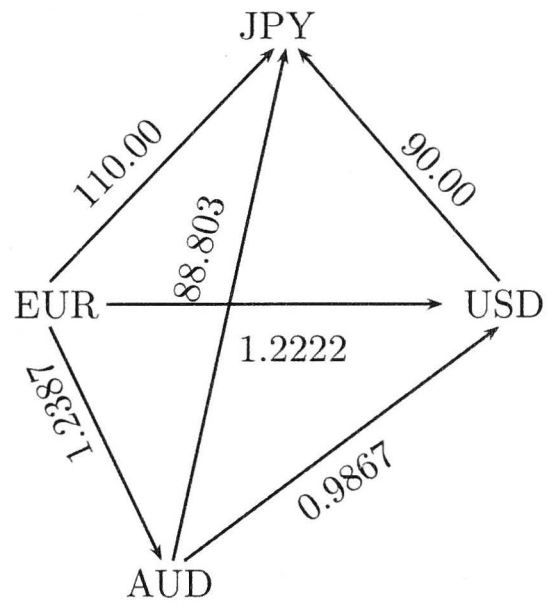

Figure 5.2: Currency tetrahedron relationship

all the currencies symmetrically with a general scheme; and c) calibrate all the coefficients simultaneously. To achieve these, De Col, Gnoatto and Grasselli [12] proposed a model in the class of multi-dimensional Heston models.

Let us suppose that there exists a universal numeraire for all currencies (e.g., gold), working as "the 0th currency" and let $S^{i/0}$ denote the ith FX rate under the 0-risk neutral measure. Hence, in the multi-dimensional Heston model, the FX rate $S^{i/0}$ is given by

$$\begin{aligned} dS_t^{i/0} &= S_t^{i/0}\left[(r_0 - r_i)dt - (\mathbf{a}^i)^T \mathrm{Diag}(\sqrt{\mathbf{v}_t})d\mathbf{Z}_t\right], \quad i = 1, \ldots, N; \\ dv_{t;k} &= \kappa_k(\theta_k - v_{t;k})dt + \xi_k\sqrt{v_{t;k}}dW_{t;k}, \quad k = 1, \ldots, d, \end{aligned}$$

where $\mathbf{Z}_t$ is a d-dimensional vector with the elements $Z_{t;k}(k = 1, 2, \ldots, d)$ and $\mathrm{Diag}(\sqrt{\mathbf{v}_t})$ denotes the diagonal matrix with the square root of the element of the d-dimensional vector $\mathbf{v}_t$ on the principle diagonal. The correlation between the Brownian motions $Z_{t;k}$ and $W_{t;k}$ is given by

$$\begin{aligned} &\text{when } i \neq j, && \begin{cases} 0 = d\langle Z_i, Z_j\rangle_t, \\ 0 = d\langle W_i, W_j\rangle_t, \\ 0 = d\langle Z_i, W_j\rangle_t, \end{cases} \\ &\text{and when } i = j = k, && \rho_k dt = d\langle Z_k, W_k\rangle_t. \end{aligned}$$

After changing to the i-risk neutral measure $\mathbb{Q}^i$ where the ith currency is deemed as the numeraire, we obtain the FX rate for the jth currency, where $j \neq i$, as

$$\begin{aligned} dS^{j/i}(t) &= S_t^{j/i}\left[(r_i - r_j)dt + (\mathbf{a}^i - \mathbf{a}^j)^T \mathrm{Diag}(\sqrt{\mathbf{v}_t})d\mathbf{Z}_t^{\mathbb{Q}^i}\right], \\ dv_{t;k} &= \kappa_k^{\mathbb{Q}^i}(\theta_k^{\mathbb{Q}^i} - v_{t;k})dt + \xi_k^{\mathbb{Q}^i}\sqrt{v_{t;k}}dW_{t;k}^{\mathbb{Q}^i}, \end{aligned} \tag{5.1}$$

where

$$\begin{aligned}
\rho_k^{\mathbb{Q}^i} &= \rho_k, \\
\xi_k^{\mathbb{Q}^i} &= \xi_k, \\
\kappa_k^{\mathbb{Q}^i} &= \kappa_k + \xi_k \rho_k a_k^i, \\
\theta_k^{\mathbb{Q}^i} &= \theta_k \frac{\kappa_k}{\kappa_k^{\mathbb{Q}^i}}.
\end{aligned}$$

Similarly, we can switch to the j-risk neutral measure $\mathbb{Q}^j$ to obtain the SDEs for the lth $(l \neq j)$ currency FX rate

$$\begin{aligned}
dS_t^{l/j} = S_t^{l/j} \left[r_j - r_l + (\mathbf{a}^j - \mathbf{a}^l)^T \mathrm{Diag}\sqrt{\mathbf{v}_t}(\mathbf{a}^j - \mathbf{a}^i) \right] dt \\
+ S_t^{l/j}(\mathbf{a}^j - \mathbf{a}^l)\mathrm{Diag}(\sqrt{\mathbf{v}_t}) d\mathbf{Z}_t^{\mathbb{Q}^j}, \qquad (5.2) \\
dv_{t;k} = \kappa_k^{\mathbb{Q}^j}(\theta_k^{\mathbb{Q}^j} - v_{t;k})dt + \xi_k^{\mathbb{Q}^j}\sqrt{v_{t;k}} dW_{t;k}^{\mathbb{Q}^j},
\end{aligned}$$

where

$$\begin{aligned}
\rho_k^{\mathbb{Q}^j} &= \rho_k^{\mathbb{Q}^i}, \\
\xi_k^{\mathbb{Q}^j} &= \xi_k^{\mathbb{Q}^i}, \\
\kappa_k^{\mathbb{Q}^j} &= \kappa_k^{\mathbb{Q}^i} + \rho_k \xi_k (a_k^j - a_k^i), \qquad (5.3) \\
\theta_k^{\mathbb{Q}^j} &= \theta_k^{\mathbb{Q}^i} \frac{\kappa_k^{\mathbb{Q}^i}}{\kappa_k^{\mathbb{Q}^j}}.
\end{aligned}$$

Hence, if we calibrate the parameters $\rho_k^{\mathbb{Q}^i}$, $\xi_k^{\mathbb{Q}^i}$, $\kappa_k^{\mathbb{Q}^i}$, $\theta_k^{\mathbb{Q}^i}$, $\mathbf{a}^i$ and $\mathbf{a}^j$ in (5.1) under the measure $\mathbb{Q}^i$, then we can obtain the parameters $\rho_k^{\mathbb{Q}^j}$, $\xi_k^{\mathbb{Q}^j}$, $\kappa_k^{\mathbb{Q}^j}$ and $\theta_k^{\mathbb{Q}^j}$ under the measure $\mathbb{Q}^j$ by (5.3) simultaneously. The parameter $\mathbf{a}^l$ can be computed numerically from the vanilla option price.

The calibration of this model involves deriving the closed formula of the vanilla option price so that the model implied volatilities can be extracted. This procedure, as said by the authors of the paper, "is quite demanding from a numerical point view". Thus, they provide an approximation for the option price and the implied volatilities. The approximation is only suitable for the options with short maturity (no more than 1 year), yet it enables a quicker calibration. The details can be found in [12].

We now return to our three currency pairs problem given above. Let $N = 3$ and $i =$ EUR, USD, JPY and $d = 2$ such that the number of parameters is 16. It is approximately equal to the parameters number (15) in three independent one-dimensional Heston models. This choice avoids over-fitting instabilities. We can

calibrate the parameters in the USD measure κ_k^{USD}, θ_k^{USD}, ξ_k^{USD}, ρ_k^{USD}, $k = 1, 2$ with the market implied volatility of EUR/USD shown in Figure 4.1. Then we can obtain the parameters for EUR/JPY under the EUR measure through (5.3). We use the results of calibration obtained in [12] as shown in Table 5.1. It can be seen from the calibration results that the Feller's condition is violated .

v_1	v_2	a_1^{USD}	a_2^{USD}	a_1^{EUR}	a_2^{EUR}	a_1^{JPY}	a_2^{JPY}
0.0137	0.0391	0.6650	1.0985	1.6177	1.3588	0.2995	1.6241
κ_1	κ_2	θ_1	θ_2	ξ_1	ξ_2	ρ_1	ρ_2
0.9418	1.7909	0.0370	0.0909	0.4912	1.0000	0.5231	-0.3980

Table 5.1: Calibration results with maturity time under 1 year.

This model is practical. Banks or financial institutes can calibrate a higher-dimensional model to the market data, then for different derivative pricing problems, the model can be reduced to lower dimension in order to avoid unnecessary computational complexity. In the thesis, we price a multi-window barrier option on one cross-currency pair, which gives us a three-dimensional Heston model.

5.2 Multi-windowed barrier option pricing

Now we compute the value of a multi-windowed barrier option on EUR/JPY with the calibration data in Table 5.1. First of all, we derive the PDE formulation from the model. We re-write (5.2) as

$$d\begin{pmatrix} S_t^{l/j} \\ v_{t;1} \\ v_{t;2} \end{pmatrix} = \boldsymbol{\mu} dt + \boldsymbol{\sigma} d\mathbf{X}_t^{\mathbb{Q}^j},$$

where $\boldsymbol{\mu}$ is a 3-dimensional vector

$$\boldsymbol{\mu} = \begin{pmatrix} S_t^{j,l}[r_j - r_l + (\mathbf{a}^j - \mathbf{a}^l)^T \mathrm{Diag}\sqrt{\mathbf{v}_t}(\mathbf{a}^j - \mathbf{a}^i)] \\ \kappa_1^{\mathbb{Q}^j}(\theta_1^{\mathbb{Q}^j} - v_{t;1}) \\ \kappa_2^{\mathbb{Q}^j}(\theta_2^{\mathbb{Q}^j} - v_{t;2}) \end{pmatrix},$$

$\boldsymbol{\sigma}$ is a 3×4 matrix

$$\boldsymbol{\sigma} = \begin{pmatrix} S_t^{l/j}(a_1^j - a_1^l)\sqrt{v_{t;1}} & 0 & S_t^{l/j}(a_2^j - a_2^l)\sqrt{v_{t;2}} & 0 \\ \xi_1\sqrt{v_{t;1}}\rho_1 & \xi_1\sqrt{v_{t;1}}\sqrt{1-\rho_1^2} & 0 & 0 \\ 0 & 0 & \xi_2\sqrt{v_{t;2}}\rho_2 & \xi_2\sqrt{v_{t;2}}\sqrt{1-\rho_2^2} \end{pmatrix},$$

and $\mathbf{X}_t^{\mathbb{Q}^j}$ is a 4-dimensional vector of $\mathbb{Q}^j$-BM with all the elements independent.

To obtain the generator $\mathcal{A}$ in Feynman-Kac's Formula, we need the matrix

$$\mathbf{M} = \frac{1}{2}\boldsymbol{\sigma}\boldsymbol{\sigma}^T = \frac{1}{2}\begin{pmatrix} \left(S^{l/j}(t)\right)^2\left[v_{t;1}\left(a_1^j - a_1^l\right)^2 + v_{t;2}(a_2^j - a_2^l)^2\right] & S_t^{l/j}v_{t;1}(a_1^j - a_1^l)\xi_1\rho_1 & S_t^{l/j}v_{t;2}(a_2^j - a_2^l)\xi_2\rho_2 \\ S_t^{l/j}v_{t;1}(a_1^j - a_1^l)\xi_1\rho_1 & (\xi_1)^2 v_{t;1} & 0 \\ S_t^{l/j}v_{t;2}(a_2^j - a_2^l)\xi_2\rho_2 & 0 & (\xi_2)^2 v_{t;2} \end{pmatrix}.$$

Then the PDE for the barrier option price $U = U(S, v_1, v_2, \tau)$ with $\tau = T - t$ is

$$\frac{\partial U}{\partial \tau} - \mathbf{V}\cdot\nabla U - (\mathbf{M}\nabla)\cdot(\nabla U) + r_j U = 0,$$

where $\mathbf{V} = \boldsymbol{\mu}$ and

$$\nabla = \begin{pmatrix} \frac{\partial}{\partial S} \\ \frac{\partial}{\partial v_1} \\ \frac{\partial}{\partial v_2} \end{pmatrix}.$$

Then we let $x = \ln S$ and change the formulation into the divergence form,

$$\frac{\partial U}{\partial \tau} - \widehat{\mathbf{V}}\cdot\nabla U - \nabla\cdot(\widehat{\mathbf{M}})\nabla U + r_j U = 0,$$

where

$$\widehat{\mathbf{V}} = \begin{pmatrix} \left[r_j - r_l + (a_1^j - a_1^l)^2\sqrt{v_1} + (a_2^j - a_2^l)^2\sqrt{v_2} - \frac{1}{2}(a_1^j - a_1^l)\xi_1\rho_1 - \frac{1}{2}(a_2^j - a_2^l)\xi_2\rho_2\right] \\ \kappa_1^{\mathbb{Q}^j}(\theta_1^{\mathbb{Q}^j} - v_1) - \frac{1}{2}\xi_1^2 \\ \kappa_2^{\mathbb{Q}^j}(\theta_2^{\mathbb{Q}^j} - v_2) - \frac{1}{2}\xi_2^2 \end{pmatrix},$$

$$\widehat{\mathbf{M}} = \frac{1}{2}\begin{pmatrix} \left[v_1\left(a_1^j - a_1^l\right)^2 + v_2(a_2^j - a_2^l)^2\right] & v_1(a_1^j - a_1^l)\xi_1\rho_1 & v_2(a_2^j - a_2^l)\xi_2\rho_2 \\ v_1(a_1^j - a_1^l)\xi_1\rho_1 & (\xi_1)^2 v_1 & 0 \\ v_2(a_2^j - a_2^l)\xi_2\rho_2 & 0 & (\xi_2)^2 v_2 \end{pmatrix},$$

and

$$\nabla = \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial v_1} \\ \frac{\partial}{\partial v_2} \end{pmatrix}.$$

We still consider the same option B as in Figure 3.5 of **Chapter 3**. Let the space domain be $x \times v_1 \times v_2$ and let $T_1 = 0.3$, $c_d = 90.0$ and $c_u = 180.0$. The

initial/boundary conditions are:

$$\text{when } \tau \in [0, 0.2], \quad \begin{cases} U = \max(e^x - K, 0), & \tau = 0, \\ U = 0, & x = \ln c_d \text{ or } x = \ln c_u, \\ \dfrac{\partial U}{\partial v_1} = 0, & v_1 = 0 \text{ or } v_1 = 5, \\ \dfrac{\partial U}{\partial v_2} = 0, & v_2 = 0 \text{ or } v_2 = 5, \end{cases}$$

$$\text{when } \tau \in [0.2, 0.5], \quad \begin{cases} U = U^*(x), & \tau = 0.2, \quad x \in [\ln c_d, \ln c_u], \\ U = 0, & \tau = 0.2, \quad x \in (-5, 7) \backslash [\ln c_d, \ln c_u], \\ U = 0, & x = -5 \text{ or } x = 7, \\ \dfrac{\partial U}{\partial v_1} = 0, & v_1 = 0 \text{ or } v_1 = 5, \\ \dfrac{\partial U}{\partial v_2} = 0, & v_2 = 0 \text{ or } v_2 = 5 \end{cases}$$

where $U^*(x)$ is the solution of U at time $T_1 = 0.3$.

When $t \in [0, T_1]$, the space domain is $[\ln c_d, \ln c_u] \times [0, 5] \times [0, 5]$. We discretise this domain with a uniform mesh where $\Delta x = \Delta v = 0.01$. Let the time step be $\Delta\tau = 0.005$. Then we obtain the solution $U^*(x)$ at T_1.

When $t \in [T_1, 0.5]$, the space domain is $[-5, 7] \times [0, 5] \times [0, 5]$. We use the non-uniform mesh where $\Delta x = \Delta v_1 = \Delta v_2 = 0.01$ when $x \in [-5, \ln c_u]$, $v_1 \in [0, 1]$ and $v_2 \in [0, 1]$, and $\Delta x = \Delta v_1 = \Delta v_2 = 0.1$ otherwise. The solution $U^*(x)$ is interpolated on this space domain as the initial condition. Let the time step be $\Delta\tau = 0.005$. We obtain the final solution of $U_{d/f}$.

We choose the volatility to be $v_1 = 0.2, v_2 = 0.3$ and plot the solution $U_{d/f}$ as a function of time to maturity t and the exchange rate of EUR/JPY in Figure 5.3.

5.3 Conclusion and extension

Firstly, we have made extension on the problem of multi-windowed barrier option pricing. In Reditus™, only Black-Scholes' model and the local volatility model are considered. In this thesis, the multi-windowed barrier options are also priced in the Heston model and the newest framework, i.e, the multi-dimensional Heston model in [12].

Secondly, we have considered the situation where the diffusion term could be degenerate in the models of Heston type. In many papers, Feller's condition is forced when the finite element method is used to price options, e.g., [52].

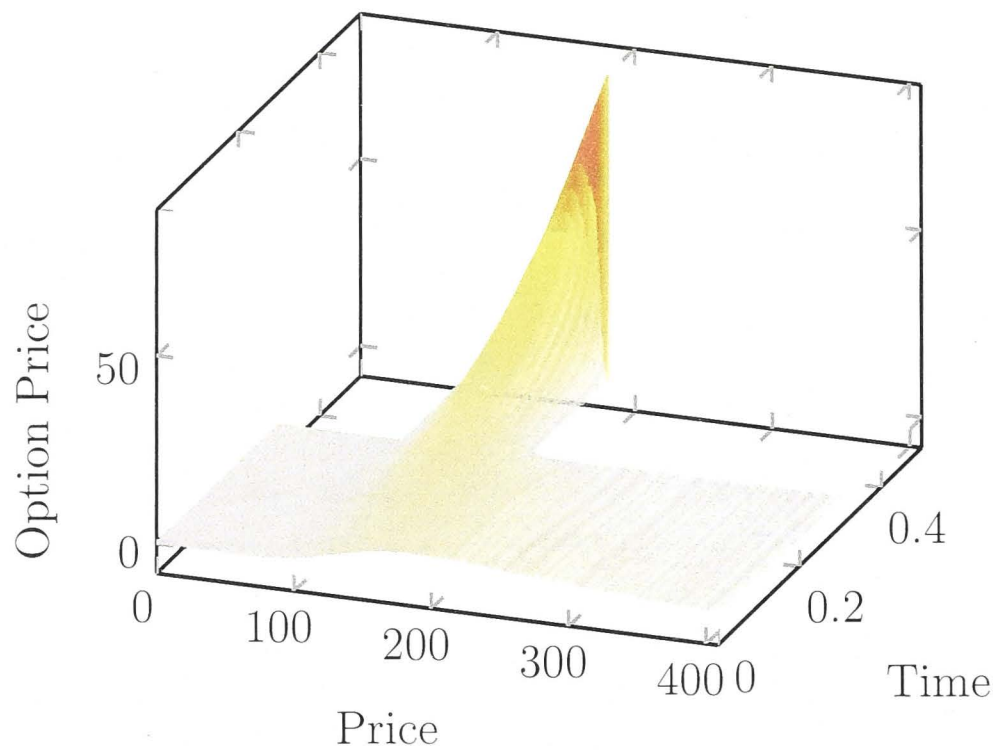

Figure 5.3: Value of the multi-window option B when $c_d = 90.0$, $c_u = 180.0$ *and* $T_1 = 0.3$. *The mesh is uniform with* $\Delta x = \Delta v_1 = \Delta v_2 = 0.01$ *when* $t \in [0.3, 0.5]$; *the mesh is non-uniform with* $\Delta x = \Delta v_1 = \Delta v_2 = 0.01$ *where* $x \in [-5, \ln c_u]$, $v_1 \in [0, 1]$ *and* $v_2 \in [0, 1]$, *and* $\Delta x = \Delta v_1 = \Delta v_2 = 0.1$ *otherwise when* $t \in [0, 0.3]$. *The time step is* $\Delta\tau = 0.005$.

Due to this requirement, a complicated scheme has to be applied to determine the boundary conditions. Based on the results of [16], the PDE approach is applicable and equivalent to the martingale approach when Feller's condition is not satisfied. Hence we have been able to determine the boundary conditions at the real boundary, i.e., the volatility term $v = 0$. Moreover, we have chosen the boundary conditions according to the properties of option, which is more practical and intuitive.

Finally, we have implemented the finite element method with C++ for Black-Scholes' model, Heston's model and the multi-dimensional Heston model, with GetFEM++ library [42] used to do the interpolation, discretization and assembly of the linear systems. The numerical results are as presented in the thesis. The C++ code is included in **Appendix B**.

There are several possibilities of further research based on the approach and the results shown in this thesis:

i The approach can be applied on other exotic options pricing, e.g., basket options and Asian options;

ii GPU and parallel algorithm can be introduced in generating meshes, assem-

bling the matrices and solving the linear system;

iii Sparse grids method could be used to achieve more efficient using of memory and computing ability, especially in the case of pricing options including more than two assets due to the higher dimensional PDEs that are generated.

Appendix A

Theorems and Lemmas

Theorem A.1 (Itô's Formula). *Suppose $X_t = (X_t^1, \ldots, X_t^d)^T$ is a d-dimensional diffusion process with SDE*

$$dX_t^i = a_t^i dt + \sum_{j=1}^{d} b_t^{i,j} dW_t^j \quad \text{where } i = 1, \ldots, d$$

and the quadratic covariation process for X_t^i and X_t^j is

$$dX_t^i dX_t^j = \sum_{k=1}^{d} b_t^{i,k} b_t^{j,k} dt.$$

Let $f(t,x) : [0,\infty) \times \mathbb{R}^d \to \mathbb{R}$ be a $C^{1,2}$ function. Then $Y_t = f(t, X_t)$ is a diffusion process with

$$dY_t = \left[\frac{\partial f}{\partial t}(t, X_t) + \sum_{i=1}^{d} \frac{\partial f}{\partial x^i}(t, X_t) a_t^i + \frac{1}{2} \frac{\partial^2 f}{\partial x^i \partial x^j} \sum_{i,j=1}^{d} \sum_{k=1}^{d} b_t^{i,k} b_t^{j,k} \right] dt + \sum_{j=1}^{d} \left(\sum_{i=1}^{n} \frac{\partial f}{\partial x^i b_t^{i,j}} \right) dW_t^j$$

Theorem A.2 (Girsanov's Theorem). *Suppose that W_t is a $\mathbb{P}$-Brownian motion with the natural filtration $\mathcal{F}_t$ and that θ_t is an $\mathcal{F}_t$-adapted process such that*

$$\mathbb{E}\left[\exp\left(\frac{1}{2} \int_0^T \theta_t^2 dt \right) \right] < \infty.$$

Then there exists a measure $\mathbb{Q}$ such that

(i) $\mathbb{Q}$ is equivalent to $\mathbb{P}$

(ii) $\frac{d\mathbb{Q}}{d\mathbb{P}} = \exp\left(-\int_0^T \theta_t dW_t - \frac{1}{2} \int_0^T \theta_t^2 dt \right)$

(iii) $\tilde{W}_t = W_t + \int_0^t \theta_s ds$ *is a* $\mathbb{Q}$*-Brownian motion*

Theorem A.3. *Let* $\langle \cdot, \cdot$ *denote the paring between* $H^*(U)$ *and* $H_0^1(U)$*.*

(i) Assume $f \in H^*(U)$*. Then there exists functions* $f^0, f^1, \ldots, f^n$ *in* $L^2(U)$ *such that*

$$\langle f, v \rangle = \int_U f^0 v + \sum_{i=1}^n f^i v_{x_i} dx \quad (v \in H_0^1(U))$$

(ii)

$$\|f\|_{H^*(U)} = \inf \left\{ \left(\int_U \sum_{i=0}^n |f^i|^2 dx \right)^{\frac{1}{2}} \right\}$$

(iii) In particular, we have

$$(v, u)_{L^2(U)} = \langle v, u \rangle$$

for all $u \in H_0^1(U)$*,* $v \in L^2(U) \subset H^*(U)$*.*

Theorem A.4. *Suppose* $\mathbf{u} \in L^2(0, T; H_0^1(U))$*, with* $\mathbf{u}' \in L^2(0, T; H^*(U))$*. Then*

(i) Then

$$\mathbf{u} \in C(0, T; L^2(U))$$

(ii) The mapping

$$t \mapsto \|\mathbf{u}(t)\|^2_{L^2(U)}$$

is absolutely continuous, with

$$\frac{d}{dt} \|\mathbf{u}\|^2_{L^2(U)} = 2\langle \mathbf{u}'(t), \mathbf{u} \rangle$$

for a.e. $0 \leq t \leq T$*.*

(iii) Furthermore, we have the estimate

$$\max_{0 \leq t \leq T} \|\mathbf{u}(t)\|^2_{L^2(U)} \leq C \left(\|\mathbf{u}\|_{L^2(0,T;H_0^1(U))} + \|\mathbf{u}'\|_{L^2(0,T;H^*(U))} \right)$$

where the constant C *depends only on* T*.*

Theorem A.5. *There exists constants* $\alpha, \beta > 0$ *and* $\gamma \geq 0$ *such that*

(i)

$$|B[u, v]| \leq \alpha \|u\|_{H_0^1(U)} \|v\|_{H_0^1(U)}$$

and

(ii)

$$\beta\|u\|^2_{H_0^1(U)} \leq B[u,u] + \gamma\|u\|^2_{L^2(U)}$$

Theorem A.6 (Gronwall's Inequality). *(i) Let η be a nonnegative, absolutely continuous function on $[0,T]$, which satisfies for a.e. t the differential inequality*

$$\eta'(t) \leq \phi(t)\eta(t) + \varphi(t)$$

where $\phi(t)$ and $\varphi(t)$ are nonnegative, summable functions on $[0,T]$. Then

$$\eta(t) \leq e^{\int_0^t \phi(s)ds}\left[\eta(0) + \int_0^t \varphi(s)ds\right]$$

for all $0 \leq t \leq T$.

(ii) In particular, if

$$\eta' \leq \phi\eta \quad \text{on } [0,T] \text{ and } \eta(0) = 0$$

then

$$\eta \equiv 0 \quad \text{on } [0,T].$$

Theorem A.7 (The Feynman-Kac Formula). *Let $\mathbf{X}_t$ be n-dimensional stochastic process satisfying the SDE*

$$d\mathbf{X}_t = \boldsymbol{\mu}(\mathbf{X}_t, t)dt + \boldsymbol{\sigma}(\mathbf{X}_t)d\mathbf{W}_t^{\mathbb{Q}}$$

where $\mathbf{W}_t^{\mathbb{Q}}$ is n-dimensional Brownian motion under the measure $\mathbb{Q}$ and let $\mathcal{A}$ be its infinitesimal generator defined by

$$\mathcal{A} = \sum_{i=1}^{n} \mu_i \frac{\partial}{\partial x_i} + \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}(\boldsymbol{\sigma}\boldsymbol{\sigma}^T)_{ij}\frac{\partial^2}{\partial x_i \partial x_j}$$

where $\mu_i = \mu_i(\mathbf{X}_t, t)$, $\sigma_i = \sigma_i(\mathbf{X}_t, t)$ and $(\boldsymbol{\sigma}\boldsymbol{\sigma}^T)_{ij}$ is element (i,j) of the matrix $\boldsymbol{\sigma}\boldsymbol{\sigma}^T$ of size $(n \times n)$.

Then

$$U(\mathbf{X}_t, t) = \mathbb{E}^{\mathbb{Q}}\left[e^{-\int_t^T r(\mathbf{X}_s, s)ds}U(\mathbf{X}_T, T)|\mathcal{F}_t\right] \quad \text{for } t < T$$

is a solution to PDE in $U(\mathbf{X}_t, t)$ given by

$$\frac{\partial U}{\partial t} + \mathcal{A}U(\mathbf{X}_t, t) - r(\mathbf{X}_t, t)U(\mathbf{X}_t, t) = 0$$

with the terminal condition $U(\mathbf{X}_T, T)$.

Appendix B

C++ Code of Implementation

B.1 Black-Scholes' model

B.1.1 Down-and-out barrier options

```
#include <iostream>
#include <string>
#include <math.h>
#include "pde_fem.h"
#include "param.h"
#include <getfem/bgeot_poly_composite.h>
#include <getfem/bgeot_comma_init.h>
#include <getfem/getfem_regular_meshes.h>
#include <getfem/getfem_derivatives.h>
#include <getfem/getfem_superlu.h>
#include <getfem/getfem_export.h>
#include "analytic.h"
//#define HEAT_EQUATION 1

using namespace std;

#ifndef HEAT_EQUATION
double coeA(const base_node &x){
        return (SIGMA*SIGMA-RISK_FREE_RATE+DIVIDEND)*x[0];
}

double coeB(const base_node &x){
        return (0.5*SIGMA*SIGMA*x[0]*x[0]);
}
#endif
```

```
double init_con(const base_node &x){
#ifndef HEAT_EQUATION
        return ((x[0]>STRIKE_PRICE)? x[0]-STRIKE_PRICE:0 );
#endif
#ifdef HEAT_EQUATION
        if(x[0] >= -3.0 && x[0] < -1.0)
                return 0;
        if(x[0] >= -1.0 && x[0] <= 1.0)
                return 1.0;
        if(x[0] > 1.0 &&  x[0] <= 3.1)
                return 0;
#endif
}

double boundary_con_vanilla(const base_node &x){
        return ((x[0]> B_LEVEL)? x[0]:0);
}


PDESolve::PDESolve(){
        dim = 1;
        c = B_LEVEL;
        k = STRIKE_PRICE;
        s = CURRENT_PRICE;
        t = MATURITY;
        btype = DOWN_OUT;
        d = DIVIDEND;
        r = RISK_FREE_RATE;
        s_step = 0.1;
        t_step = 0.025;
        residual = 1E-10;
        it_number = 0;
        mymesh = new getfem::mesh;
        mfu = new getfem::mesh_fem;
        mf_coe = new getfem::mesh_fem;
        mim = new getfem::mesh_im;
}

PDESolve::~PDESolve(){
        delete mymesh;
        delete mfu;
        delete mf_coe;
        delete mim;
}
```

```
void PDESolve::setBarrierType(int b_type){
        btype = b_type;
};

void PDESolve::setVolatility(double volatility){
        sigma = volatility;
};

void PDESolve::setRiskFreeRate(double rate){
        r = rate;
};

void PDESolve::setStrike(double strike){
        k = strike;
};

void PDESolve::setBarrierLevel(double b_level){
        c = b_level;
};

void PDESolve::setMaturity(double maturity){
        t = maturity;
};

void PDESolve::setCurrentPrice(double price){
        s = price;
};

void PDESolve::setDividend(double dividend){
        d = dividend;
};

void PDESolve::setSpaceStep(double step){
        s_step = step;
};


void PDESolve::BuildMesh(double start, double end){
        int i = 0, j=0;
        switch(btype){
                case DOWN_OUT:
                #ifndef HEAT_EQUATION
                if(start == 0 && end == 0)
```

```
{
        start_p = c;
        it_number = 10000;
}
else{
        start_p = start;
        it_number = (end-start_p)/s_step;

}
#endif
#ifdef HEAT_EQUATION
        start_p = start;
        it_number = (end-start_p)/s_step;
    #endif
    break;
    case UP_OUT:
        start_p = 0;
        end_p = c;
    break;
    case DOWN_IN:
    break;
    case UP_IN:
    break;
    default:
    break;
}

bgeot::base_node org(dim);
std::vector<bgeot::base_small_vector> vect(dim);
bgeot::base_small_vector tmp(dim);
std::vector<int> ref(dim);
//here is a work-around for 1-dim case 'cause the it
    considers 1 parameter as dimension but not the
    coordinate
//while this form fits the high dimensional case
    better than the given example
for(i=0; i<dim; i++)
{
        org[i] = start_p;
        for(j=0; j< dim; j++)
            tmp[i] = s_step;
        vect[i] = tmp;
        ref[i]= it_number;
}
```

```
        mymesh->clear();
        getfem::parallelepiped_regular_simplex_mesh(*mymesh,
            dim, org, vect.begin(), ref.begin());
}

void PDESolve::BuildFEM(bool init){
        mfu->init_with_mesh(*mymesh);
        mim->init_with_mesh(*mymesh);
        mf_coe->init_with_mesh(*mymesh);
        mfu->set_classical_finite_element(1);
        mf_coe->set_classical_finite_element(1);
        mim->set_integration_method(getfem::
            int_method_descriptor("IM_EXACT_SIMPLEX(1)"));
}

void PDESolve::AsmGradBaseMatrix(sparse_matrix_type &M,
                const getfem::mesh_im &mim,
                const getfem::mesh_fem &mf,
                const getfem::mesh_fem &mfdata,
                const plain_vector &V){

        getfem::generic_assembly assem;
        assem.push_mi(mim);
        assem.push_mf(mf);
        assem.push_mf(mfdata);
        assem.push_data(V);
        assem.push_mat(M);
        if(mf.get_qdim() == 1)
                assem.set("a=data$1(#2);"
                        "M$1(#1,#1)+=comp(Base(#1).Grad(#1).
                            Base(#2))(:,:,j,i).a(i)");
        else
                assem.set("a=data$1(#2);"
                        "M$1(#1,#1)+=comp(vGrad(#1).vBase
                            (#1).Base(#2))(:,j,k,:,j,p).a(p)"
                            );
        assem.assembly();

}

void PDESolve::AsmGradGradMatrix(sparse_matrix_type &M,
                const getfem::mesh_im &mim,
                const getfem::mesh_fem &mf,
```

```
        const getfem::mesh_fem &mfdata,
        const plain_vector &V){
getfem::generic_assembly assem;
assem.push_mi(mim);
assem.push_mf(mf);
assem.push_mf(mfdata);
assem.push_data(V);
assem.push_mat(M);
if(mf.get_qdim() == 1)
        assem.set("a=data$1(#2);"
                "M$1(#1,#1)+=comp(Grad(#1).Grad(#1).
                    Base(#2))(:,j,:,j,i).a(i)");
else
                assem.set("a=data$1(#2);"
                        "M$1(#1,#1)+=sym(comp(vGrad(#1).
                            vGrad(#1).vBase(#2))(:,j,k,:,j,k
                            ,:,p).a(p))");
        assem.assembly();

}
void PDESolve::GetInitU(bool init){
        int nb_dof = 0, coe_nb_dof = 0;
        int nb_col = 0;
        nb_dof = mfu->nb_dof();
        coe_nb_dof = mf_coe->nb_dof();
        plain_vector vT(coe_nb_dof);
        sparse_matrix_type mH(nb_dof, nb_dof);
        plain_vector vR(nb_dof);
        plain_vector vTR(nb_dof);

        if(init)
        {
        getfem::interpolation_function(*mf_coe, vT,
            init_con);
}
else
{
        getfem::interpolation(*mfu_o, *mfu, vFU, vT,
            0);
        vT[0] = 0;
        vT[coe_nb_dof-1] = 0;
}
//derive the linear system HU = R
asm_mass_matrix(mH, *mim, *mfu);
```

```
        asm_source_term(vR, *mim, *mfu, *mf_coe, vT);
        //apply the boundary condition
        FindBoundary();
        if(init)
                nb_col = ApplyBoundary(0, 0);
        else
                nb_col = ApplyBoundary(0, 120.0);
        gmm::resize(mNS, nb_dof ,nb_col);
        gmm::mult(mH, vTU, gmm::scaled(vR, -1.0), vTR);
        gmm::resize(vU, nb_col);
        gmm::clear(vU);
        gmm::resize(vR, nb_col);

        gmm::mult(gmm::transposed(mNS), gmm::scaled(vTR, -1.0)
            , vR);
        sparse_matrix_type mTH(nb_col, nb_dof);
        gmm::mult(gmm::transposed(mNS), mH, mTH);
        gmm::resize(mH, nb_col, nb_col);
        sparse_matrix_type mTNS(nb_dof, nb_col);
        gmm::copy(mNS, mTNS);
        gmm::mult(mTH, mTNS, mH);

        //solving the system to get U at time 0
        gmm::iteration iter(residual, 1, 40000);
        gmm::ilut_precond<sparse_matrix_type> P(mH, 50, 1E-9);
        gmm::cg(mH, vU, vR, P, iter);
}

void PDESolve::AssembleMatrix(){
        int nb_dof = 0, coe_nb_dof = 0;
        int i = 0;

        coe_nb_dof = mf_coe->nb_dof();
        nb_dof = mfu->nb_dof();
        plain_vector vA(coe_nb_dof), vB(coe_nb_dof), vT(
            coe_nb_dof);
        #ifndef HEAT_EQUATION
        getfem::interpolation_function(*mf_coe, vA, coeA);
        getfem::interpolation_function(*mf_coe, vB, coeB);
        #endif
        //assembly \int_{\Omega}w^{i}w^{j}
        gmm::resize(mM, nb_dof, nb_dof);
        gmm::clear(mM);
        asm_mass_matrix(mM, *mim, *mfu, *mf_coe);
```

```
        //assembly \int_{\Omega}(\sigma^{2}-r+d)
        //                                    Sw^{i}w_{s}^{j}
        gmm::resize(mA, nb_dof, nb_dof);
        gmm::clear(mA);
        #ifndef HEAT_EQUATION
        AsmGradBaseMatrix(mA, *mim, *mfu, *mf_coe, vA);
        #endif
        //assembly \int_{\Omega}(1/2 \sigma^{2}S^{2}w^{j}_{s}w
            ^{i}_{s})
        gmm::resize(mS, nb_dof, nb_dof);
        gmm::clear(mS);
        #ifndef HEAT_EQUATION
        asm_stiffness_matrix_for_laplacian(mS, *mim, *mfu,
                *mf_coe, vB);
        #endif
        #ifdef HEAT_EQUATION
        asm_stiffness_matrix_for_homogeneous_laplacian(mS,
                *mim, *mfu);
        #endif

}

void PDESolve::FindBoundary(){

        getfem::mesh_region border_faces;
        getfem::outer_faces_of_mesh(*mymesh, border_faces);

        for (getfem::mr_visitor i(border_faces); !i.finished()
            ; ++i) {
                assert(i.is_face());
                base_node un = mymesh->
                    normal_of_face_of_convex(i.cv(), i.f());
                un /= gmm::vect_norm2(un);
                mymesh->region(DIRICHLET_BOUNDARY).add(i.cv(),
                    i.f());
        }
}

int PDESolve::ApplyBoundary(double time, double up){


        int nb_dof = 0, coe_nb_dof = 0;
        nb_dof = mfu->nb_dof();
        coe_nb_dof = mf_coe->nb_dof();
```

```
        gmm::resize(vTU, nb_dof);
        gmm::clear(vTU);
        gmm::resize(mNS, nb_dof, nb_dof);
        gmm::clear(mNS);
        plain_vector vT(coe_nb_dof);
        gmm::clear(vT);
        plain_vector vR(nb_dof);
        col_sparse_matrix_type mH(nb_dof, nb_dof);
        plain_vector vK(coe_nb_dof);
        gmm::clear(vK);

        #ifndef HEAT_EQUATION
        if(up == 0.0)//up boundary = 0 means it is infinite
        {
                getfem::interpolation_function(*mf_coe, vT,
                   boundary_con_vanilla);
                for(int j=0; j< coe_nb_dof; j++)
                {
                        vK[j] = -1*k*exp(-1*r*time);
                }
                gmm::add(vK, vT);
                vT[0] = 0;
        }
        else{
                gmm::clear(vT);
        }
        #endif
        getfem::asm_dirichlet_constraints(mH, vR, *mim, *mfu,
           *mf_coe, *mf_coe, vT, DIRICHLET_BOUNDARY);
        gmm::clean(mH, 1.0E-12);
        return getfem::Dirichlet_nullspace(mH, mNS, vR, vTU);

}

double PDESolve::solve(){
        double o_price = 0.0;
        int nb_dof = 0, nb_col = 0;
        int i = 0, converge=0;
        double t_time = 0.0;
        int period = 2;

        char s[100];
        gmm::iteration iter(residual, 1, 40000);
        double time = dal::uclock_sec();
```

```
BuildMesh(0, 0);
BuildFEM(true);
AssembleMatrix();

nb_dof = mfu->nb_dof();


//get u at time 0 from initial condition
GetInitU(true);

nb_col = gmm::vect_size(vU);
gmm::resize(vFU, nb_dof);
gmm::clear(vFU);

gmm::clean(vU, 1.0E-12);
gmm::mult(mNS, vU, vTU, vFU);
gmm::clean(vFU, 1.0E-12);

//export the result as dx
getfem::dx_export exp("solution.dx", true);
exp.exporting(*mfu);

exp.write_point_data(*mfu, vFU);
exp.serie_add_object("option_price");

//derive the linear system for time 0
gmm::add(mS, mA);
#ifndef HEAT_EQUATION
gmm::add(gmm::scaled(mM,r), mA);
#endif
gmm::scale(mM, 1/t_step);
gmm::add(mM, mA);

//store the final result with boundary condition
plain_vector vFUt(nb_dof);
gmm::clear(vFUt);

plain_vector vT(nb_col);
gmm::clear(vT);

plain_vector vOTU(nb_dof);
gmm::clear(vOTU);
```

```
        sparse_matrix_type mTA(nb_col, nb_dof);
        sparse_matrix_type mOA(nb_dof, nb_dof);
        sparse_matrix_type mOM(nb_dof, nb_dof);
        sparse_matrix_type mTM(nb_col, nb_dof);
        sparse_matrix_type mTNS(nb_dof, nb_col);
        gmm::copy(mA, mOA);
        gmm::copy(mM, mOM);
        gmm::resize(mA, nb_col, nb_col);
        gmm::resize(mM, nb_col, nb_col);

        gmm::ilut_precond<sparse_matrix_type> P;

        for(i=0; i< (t-0.2)/t_step; i++){

                t_time = t_step*(i+1);
                ApplyBoundary(t_time, 0);
                gmm::resize(mNS, nb_dof ,nb_col);

                gmm::mult(gmm::transposed(mNS), mOA, mTA);
                gmm::mult(gmm::transposed(mNS), mOM, mTM);
                gmm::copy(mNS, mTNS);
                gmm::mult(mTA, mTNS, mA);
                gmm::mult(mTM, mTNS, mM);
                gmm::mult(mOA, vTU, vOTU);
                gmm::mult(mOM, vFU, gmm::scaled(vOTU, -1),
                    vFUt);
                gmm::mult(gmm::transposed(mNS), vFUt, vT);

                P.build_with(mA, 50, 1E-9);
                iter.init();
                iter.set_rhsnorm(residual);
                iter.set_noisy(1);
                iter.set_maxiter(40000);
                gmm::gmres(mA, vU, vT, P, 50, iter);
                converge = iter.converged();

                gmm::clean(vU, 1.0E-12);
                gmm::mult(mNS, vU, vTU, vFU);
                gmm::clean(vFU, 1.0E-12);
                exp.write_point_data(*mfu, vFU);
                exp.serie_add_object("option_price");

        }
```

```
//next period
t_step = 0.01;
s_step = 0.05;
mymesh_o = mymesh;
mymesh = new getfem::mesh;
BuildMesh(80.0, 120);
mfu_o = mfu;
mf_coe_o = mf_coe;
mf_coe = new getfem::mesh_fem;
mfu = new getfem::mesh_fem;
mim_o = mim;
mim = new getfem::mesh_im;
BuildFEM(false);
AssembleMatrix();
//get u at time t+1 from time t

nb_dof = mfu->nb_dof();
GetInitU(false);
mymesh_o->write_to_file("bs_mesh_o.msh");
delete mymesh_o;
delete mfu_o;
delete mf_coe_o;
delete mim_o;
mymesh->write_to_file("bs_mesh.msh");

gmm::add(mS, mA);
#ifndef HEAT_EQUATION
gmm::add(gmm::scaled(mM,r), mA);
#endif
gmm::scale(mM, 1/t_step);
gmm::add(mM, mA);

nb_col = gmm::vect_size(vU);
gmm::resize(mTA, nb_col, nb_dof);
gmm::resize(mOA, nb_dof, nb_dof);
gmm::resize(mOM, nb_dof, nb_dof);
gmm::resize(mTM, nb_col, nb_dof);
gmm::copy(mA, mOA);
gmm::copy(mM, mOM);
gmm::resize(mA, nb_col, nb_col);
gmm::resize(mM, nb_col, nb_col);


gmm::resize (mTNS, nb_dof, nb_col);
```

```
getfem::dx_export expn("solution_1.dx", true);
expn.exporting(*mfu);


gmm::ilut_precond<sparse_matrix_type> PN(mA, 50,
                                         1E-9);

//store the final result with boundary condition
gmm::resize(vFUt, nb_dof);
gmm::resize(vFU, nb_dof);
gmm::clear(vFU);
gmm::clear(vFUt);

gmm::resize(vOTU ,nb_dof);
gmm::clear(vOTU);

gmm::resize (vT, nb_col);
gmm::clear(vT);

gmm::clean(vU, 1.0E-12);
gmm::mult(mNS, vU, vTU, vFU);
gmm::clean(vFU, 1.0E-12);
expn.write_point_data(*mfu, vFU);
expn.serie_add_object("option_price");

for(i=0; i< 0.2/t_step; i++){

        t_time = t_step*(i+1);
        ApplyBoundary(t_time, 120.0);
        gmm::resize(mNS, nb_dof ,nb_col);

        gmm::mult(gmm::transposed(mNS), mOA, mTA);
        gmm::mult(gmm::transposed(mNS), mOM, mTM);
        gmm::copy(mNS, mTNS);
        gmm::mult(mTA, mTNS, mA);
        gmm::mult(mTM, mTNS, mM);

        gmm::mult(mOA, vTU, vOTU);
        gmm::mult(mOM, vFU, gmm::scaled(vOTU, -1),
            vFUt);
        gmm::mult(gmm::transposed(mNS), vFUt, vT);
```

```
        P.build_with(mA, 50, 1E-9);
        iter.init();
        iter.set_rhsnorm(residual);
        iter.set_noisy(1);
        iter.set_maxiter(40000);
        gmm::gmres(mA, vU, vT, P, 50, iter);
        converge = iter.converged();

        gmm::clean(vU, 1.0E-12);
        gmm::mult(mNS, vU, vTU, vFU);
        gmm::clean(vFU, 1.0E-12);
        expn.write_point_data(*mfu, vFU);
        expn.serie_add_object("option_price");

    }

    return converge;
}
```

B.1.2 Multi-windowed barrier options

Option A

```
#include <iostream>
#include <string>
#include <math.h>
#include "pde_fem.h"
#include "param.h"
#include <getfem/bgeot_poly_composite.h>
#include <getfem/bgeot_comma_init.h>
#include <getfem/getfem_regular_meshes.h>
#include <getfem/getfem_derivatives.h>
#include <getfem/getfem_superlu.h>
#include <getfem/getfem_export.h>
#include "analytic.h"
//#define HEAT_EQUATION 1

using namespace std;

#ifndef HEAT_EQUATION
double coeA(const base_node &x){
        return (SIGMA*SIGMA-RISK_FREE_RATE+DIVIDEND)*x[0];
}

double coeB(const base_node &x){
```

```
        return (0.5*SIGMA*SIGMA*x[0]*x[0]);
}
#endif

double init_con(const base_node &x){
#ifndef HEAT_EQUATION
        return ((x[0]>STRIKE_PRICE)? x[0]-STRIKE_PRICE:0 );
#endif
#ifdef HEAT_EQUATION
        if(x[0] >= -3.0 && x[0] < -1.0)
                return 0;
        if(x[0] >= -1.0 && x[0] <= 1.0)
                return 1.0;
        if(x[0] > 1.0 &&  x[0] <= 3.1)
                return 0;
#endif
}

double boundary_con_vanilla(const base_node &x){
        return ((x[0]> B_LEVEL)? x[0]:0);
}


PDESolve::PDESolve(){
        dim = 1;
        c = B_LEVEL;
        k = STRIKE_PRICE;
        s = CURRENT_PRICE;
        t = MATURITY;
        btype = DOWN_OUT;
        d = DIVIDEND;
        r = RISK_FREE_RATE;
        s_step = 0.1;
        t_step = 0.025;
        residual = 1E-10;
        it_number = 0;
        mymesh = new getfem::mesh;
        mfu = new getfem::mesh_fem;
        mf_coe = new getfem::mesh_fem;
        mim = new getfem::mesh_im;
}

PDESolve::~PDESolve(){
        delete mymesh;
```

```
        delete mfu;
        delete mf_coe;
        delete mim;
}

void PDESolve::setBarrierType(int b_type){
        btype = b_type;
};

void PDESolve::setVolatility(double volatility){
        sigma = volatility;
};

void PDESolve::setRiskFreeRate(double rate){
        r = rate;
};

void PDESolve::setStrike(double strike){
        k = strike;
};

void PDESolve::setBarrierLevel(double b_level){
        c = b_level;
};

void PDESolve::setMaturity(double maturity){
        t = maturity;
};

void PDESolve::setCurrentPrice(double price){
        s = price;
};

void PDESolve::setDividend(double dividend){
        d = dividend;
};

void PDESolve::setSpaceStep(double step){
        s_step = step;
};


void PDESolve::BuildMesh(double start, double end){
        int i = 0, j=0;
```

```
        switch(btype){
                case DOWN_OUT:
                #ifndef HEAT_EQUATION
                if(start == 0 && end == 0)
                {
                        start_p = c;
                        it_number = 10000;
                }
                else{
                        start_p = start;
                        it_number = (end-start_p)/s_step;

                }
                #endif
                #ifdef HEAT_EQUATION
                        start_p = start;
                        it_number = (end-start_p)/s_step;
                #endif
                break;
                case UP_OUT:
                        start_p = 0;
                        end_p = c;
                break;
                case DOWN_IN:
                break;
                case UP_IN:
                break;
                default:
                break;
        }

        bgeot::base_node org(dim);
        std::vector<bgeot::base_small_vector> vect(dim);
        bgeot::base_small_vector tmp(dim);
        std::vector<int> ref(dim);
        //here is a work-around for 1-dim case 'cause the it
            considers 1 parameter as dimension but not the
            coordinate
        //while this form fits the high dimensional case
            better than the given example
        for(i=0; i<dim; i++)
        {
                org[i] = start_p;
                for(j=0; j< dim; j++)
```

```
                        tmp[i] = s_step;
                vect[i] = tmp;
                ref[i]= it_number;
        }

        mymesh->clear();
        getfem::parallelepiped_regular_simplex_mesh(*mymesh,
            dim, org,
                vect.begin(), ref.begin());
}

void PDESolve::BuildFEM(bool init){
        mfu->init_with_mesh(*mymesh);
        mim->init_with_mesh(*mymesh);
        mf_coe->init_with_mesh(*mymesh);
        mfu->set_classical_finite_element(1);
        mf_coe->set_classical_finite_element(1);
        mim->set_integration_method(getfem::
            int_method_descriptor
                ("IM_EXACT_SIMPLEX(1)"));
}

void PDESolve::AsmGradBaseMatrix(sparse_matrix_type &M,
                const getfem::mesh_im &mim,
                const getfem::mesh_fem &mf,
                const getfem::mesh_fem &mfdata,
                const plain_vector &V){

        getfem::generic_assembly assem;
        assem.push_mi(mim);
        assem.push_mf(mf);
        assem.push_mf(mfdata);
        assem.push_data(V);
        assem.push_mat(M);
        if(mf.get_qdim() == 1)
                assem.set("a=data$1(#2);"
                        "M$1(#1,#1)+=comp(Base(#1).Grad(#1).
                            Base(#2))"
                        "(:,:,j,i).a(i)");
        else
                assem.set("a=data$1(#2);"
                        "M$1(#1,#1)+=comp(vGrad(#1).vBase
                            (#1).Base(#2))"
                        "(:,j,k,:,j,p).a(p)");
```

```
        assem.assembly();

}

void PDESolve::AsmGradGradMatrix(sparse_matrix_type &M,
                const getfem::mesh_im &mim,
                const getfem::mesh_fem &mf,
                const getfem::mesh_fem &mfdata,
                const plain_vector &V){
        getfem::generic_assembly assem;
        assem.push_mi(mim);
        assem.push_mf(mf);
        assem.push_mf(mfdata);
        assem.push_data(V);
        assem.push_mat(M);
        if(mf.get_qdim() == 1)
                assem.set("a=data$1(#2);"
                        "M$1(#1,#1)+=comp(Grad(#1).Grad(#1).
                           Base(#2))"
                        "(:,j,:,j,i).a(i)");
        else
                assem.set("a=data$1(#2);"
                        "M$1(#1,#1)+=sym(comp(vGrad(#1).
                           vGrad(#1).vBase(#2))"
                        "(:,j,k,:,j,k,:,p).a(p))");
        assem.assembly();

}
void PDESolve::GetInitU(bool init){
        int nb_dof = 0, coe_nb_dof = 0;
        int nb_col = 0;
        nb_dof = mfu->nb_dof();
        coe_nb_dof = mf_coe->nb_dof();
        plain_vector vT(coe_nb_dof);
        sparse_matrix_type mH(nb_dof, nb_dof);
        plain_vector vR(nb_dof);
        plain_vector vTR(nb_dof);

        if(init)
        {
                getfem::interpolation_function(*mf_coe, vT,
                   init_con);
                vT[coe_nb_dof-1] = 0;
        }
```

```
        else
        {
                getfem::interpolation(*mfu_o, *mfu, vFU, vT,
                    0);
        }
        //derive the linear system HU = R
        asm_mass_matrix(mH, *mim, *mfu);
        asm_source_term(vR, *mim, *mfu, *mf_coe, vT);
        //apply the boundary condition
        FindBoundary();
        if(init)
                nb_col = ApplyBoundary(0, 120.0);
        else
                nb_col = ApplyBoundary(0, 120.0);
        gmm::resize(mNS, nb_dof ,nb_col);
        gmm::mult(mH, vTU, gmm::scaled(vR, -1.0), vTR);
        gmm::resize(vU, nb_col);
        gmm::clear(vU);
        gmm::resize(vR, nb_col);

        gmm::mult(gmm::transposed(mNS),
                gmm::scaled(vTR, -1.0), vR);
        sparse_matrix_type mTH(nb_col, nb_dof);
        gmm::mult(gmm::transposed(mNS), mH, mTH);
        gmm::resize(mH, nb_col, nb_col);
        sparse_matrix_type mTNS(nb_dof, nb_col);
        gmm::copy(mNS, mTNS);
        gmm::mult(mTH, mTNS, mH);

        //solving the system to get U at time 0
        gmm::iteration iter(residual, 1, 40000);
        gmm::ilut_precond<sparse_matrix_type> P(mH, 50, 1E-9);
        gmm::cg(mH, vU, vR, P, iter);
}

void PDESolve::AssembleMatrix(){
        int nb_dof = 0, coe_nb_dof = 0;
        int i = 0;

        coe_nb_dof = mf_coe->nb_dof();
        nb_dof = mfu->nb_dof();
        plain_vector vA(coe_nb_dof), vB(coe_nb_dof), vT(
            coe_nb_dof);
        #ifndef HEAT_EQUATION
```

```
        getfem::interpolation_function(*mf_coe, vA, coeA);
        getfem::interpolation_function(*mf_coe, vB, coeB);
        #endif
        //assembly \int_{\Omega}w^{i}w^{j}
        gmm::resize(mM, nb_dof, nb_dof);
        gmm::clear(mM);
        asm_mass_matrix(mM, *mim, *mfu, *mf_coe);
        //assembly \int_{\Omega}(\sigma^{2}-r+d)
        //                                    Sw^{i}w_{s}^{j}
        gmm::resize(mA, nb_dof, nb_dof);
        gmm::clear(mA);
        #ifndef HEAT_EQUATION
        AsmGradBaseMatrix(mA, *mim, *mfu, *mf_coe, vA);
        #endif
        //assembly \int_{\Omega} (1/2 \sigma^{2}S^{2}w^{j}_{s}
            w^{i}_{s})
        gmm::resize(mS, nb_dof, nb_dof);
        gmm::clear(mS);
        #ifndef HEAT_EQUATION
        asm_stiffness_matrix_for_laplacian(mS, *mim, *mfu, *
            mf_coe, vB);
        #endif
        #ifdef HEAT_EQUATION
        asm_stiffness_matrix_for_homogeneous_laplacian(mS, *
            mim, *mfu);
        #endif

}

void PDESolve::FindBoundary(){

        getfem::mesh_region border_faces;
        getfem::outer_faces_of_mesh(*mymesh, border_faces);

        for (getfem::mr_visitor i(border_faces); !i.finished()
            ; ++i) {
                assert(i.is_face());
                base_node un = mymesh->
                    normal_of_face_of_convex(i.cv(), i.f());
                un /= gmm::vect_norm2(un);
                mymesh->region(DIRICHLET_BOUNDARY).add(i.cv(),
                     i.f());
        }
}
```

```
int PDESolve::ApplyBoundary(double time, double up){


        int nb_dof = 0, coe_nb_dof = 0;
        nb_dof = mfu->nb_dof();
        coe_nb_dof = mf_coe->nb_dof();
        gmm::resize(vTU, nb_dof);
        gmm::clear(vTU);
        gmm::resize(mNS, nb_dof, nb_dof);
        gmm::clear(mNS);
        plain_vector vT(coe_nb_dof);
        gmm::clear(vT);
        plain_vector vR(nb_dof);
        col_sparse_matrix_type mH(nb_dof, nb_dof);
        plain_vector vK(coe_nb_dof);
        gmm::clear(vK);

        #ifndef HEAT_EQUATION
        if(up == 0.0)//up boundary = 0 means it is infinite
        {
                getfem::interpolation_function(*mf_coe, vT,
                   boundary_con_vanilla);
                for(int j=0; j< coe_nb_dof; j++)
                {
                        vK[j] = -1*k*exp(-1*r*time);
                }
                gmm::add(vK, vT);
                vT[0] = 0;
        }
        else{
                gmm::clear(vT);
        }
        #endif
        getfem::asm_dirichlet_constraints(mH, vR, *mim, *mfu,
                *mf_coe, *mf_coe, vT, DIRICHLET_BOUNDARY);
        gmm::clean(mH, 1.0E-12);
        return getfem::Dirichlet_nullspace(mH, mNS, vR, vTU);

}

double PDESolve::solve(){
        double o_price = 0.0;
        int nb_dof = 0, nb_col = 0;
```

```
int i = 0, converge=0;
double t_time = 0.0;
int period = 2;

char s[100];
gmm::iteration iter(residual, 1, 40000);
double time = dal::uclock_sec();

t_step = 0.01;
s_step = 0.05;

BuildMesh(80.0, 120.0);
BuildFEM(true);
AssembleMatrix();

nb_dof = mfu->nb_dof();


//get u at time 0 from initial condition
GetInitU(true);

nb_col = gmm::vect_size(vU);
gmm::resize(vFU, nb_dof);
gmm::clear(vFU);

gmm::clean(vU, 1.0E-12);
gmm::mult(mNS, vU, vTU, vFU);
gmm::clean(vFU, 1.0E-12);

//export the result as dx
getfem::dx_export exp("solution.dx", true);
exp.exporting(*mfu);

exp.write_point_data(*mfu, vFU);
exp.serie_add_object("option_price");

//derive the linear system for time 0
gmm::add(mS, mA);
#ifndef HEAT_EQUATION
gmm::add(gmm::scaled(mM,r), mA);
#endif
gmm::scale(mM, 1/t_step);
gmm::add(mM, mA);
```

```
//store the final result with boundary condition
plain_vector vFUt(nb_dof);
gmm::clear(vFUt);

plain_vector vT(nb_col);
gmm::clear(vT);

plain_vector vOTU(nb_dof);
gmm::clear(vOTU);

sparse_matrix_type mTA(nb_col, nb_dof);
sparse_matrix_type mOA(nb_dof, nb_dof);
sparse_matrix_type mOM(nb_dof, nb_dof);
sparse_matrix_type mTM(nb_col, nb_dof);
sparse_matrix_type mTNS(nb_dof, nb_col);
gmm::copy(mA, mOA);
gmm::copy(mM, mOM);
gmm::resize(mA, nb_col, nb_col);
gmm::resize(mM, nb_col, nb_col);

gmm::ilut_precond<sparse_matrix_type> P;

for(i=0; i< (t-0.2)/t_step; i++){

        t_time = t_step*(i+1);
        ApplyBoundary(t_time, 120.0);
        gmm::resize(mNS, nb_dof ,nb_col);

        gmm::mult(gmm::transposed(mNS), mOA, mTA);
        gmm::mult(gmm::transposed(mNS), mOM, mTM);
        gmm::copy(mNS, mTNS);
        gmm::mult(mTA, mTNS, mA);
        gmm::mult(mTM, mTNS, mM);
        gmm::mult(mOA, vTU, vOTU);
        gmm::mult(mOM, vFU, gmm::scaled(vOTU, -1),
            vFUt);
        gmm::mult(gmm::transposed(mNS), vFUt, vT);

        P.build_with(mA, 50, 1E-9);
        iter.init();
        iter.set_rhsnorm(residual);
        iter.set_noisy(1);
        iter.set_maxiter(40000);
        gmm::gmres(mA, vU, vT, P, 50, iter);
```

```
            converge = iter.converged();

            gmm::clean(vU, 1.0E-12);
            gmm::mult(mNS, vU, vTU, vFU);
            gmm::clean(vFU, 1.0E-12);
            cout << "vU is " << vU << endl;
            cout << "result is " << vFU << endl;
            exp.write_point_data(*mfu, vFU);
            exp.serie_add_object("option_price");

        }

        //next period
        t_step = 0.025;
        s_step = 0.1;
        mymesh_o = mymesh;
        mymesh = new getfem::mesh;
        BuildMesh(0, 0);
        mfu_o = mfu;
        mf_coe_o = mf_coe;
        mf_coe = new getfem::mesh_fem;
        mfu = new getfem::mesh_fem;
        mim_o = mim;
        mim = new getfem::mesh_im;
        BuildFEM(false);
        AssembleMatrix();
        //get u at time t+1 from time t

        nb_dof = mfu->nb_dof();
        GetInitU(false);
        mymesh_o->write_to_file("bs_mesh_o.msh");
        delete mymesh_o;
        delete mfu_o;
        delete mf_coe_o;
        delete mim_o;
        mymesh->write_to_file("bs_mesh.msh");

        gmm::add(mS, mA);
        #ifndef HEAT_EQUATION
        gmm::add(gmm::scaled(mM,r), mA);
        #endif
        gmm::scale(mM, 1/t_step);
        gmm::add(mM, mA);
```

```
nb_col = gmm::vect_size(vU);
gmm::resize(mTA, nb_col, nb_dof);
gmm::resize(mOA, nb_dof, nb_dof);
gmm::resize(mOM, nb_dof, nb_dof);
gmm::resize(mTM, nb_col, nb_dof);
gmm::copy(mA, mOA);
gmm::copy(mM, mOM);
gmm::resize(mA, nb_col, nb_col);
gmm::resize(mM, nb_col, nb_col);


gmm::resize (mTNS, nb_dof, nb_col);


getfem::dx_export expn("solution_1.dx", true);
expn.exporting(*mfu);


gmm::ilut_precond<sparse_matrix_type> PN(mA, 50, 1E-9)
    ;

//store the final result with boundary condition
gmm::resize(vFUt, nb_dof);
gmm::resize(vFU, nb_dof);
gmm::clear(vFU);
gmm::clear(vFUt);

gmm::resize(vOTU ,nb_dof);
gmm::clear(vOTU);

gmm::resize (vT, nb_col);
gmm::clear(vT);

gmm::clean(vU, 1.0E-12);
gmm::mult(mNS, vU, vTU, vFU);
gmm::clean(vFU, 1.0E-12);
expn.write_point_data(*mfu, vFU);
expn.serie_add_object("option_price");

for(i=0; i< 0.2/t_step; i++){

        t_time = t_step*(i+1);
        ApplyBoundary(t_time, 120.0);
        gmm::resize(mNS, nb_dof ,nb_col);
```

```
        gmm::mult(gmm::transposed(mNS), mOA, mTA);
        gmm::mult(gmm::transposed(mNS), mOM, mTM);
        gmm::copy(mNS, mTNS);
        gmm::mult(mTA, mTNS, mA);
        gmm::mult(mTM, mTNS, mM);

        gmm::mult(mOA, vTU, vOTU);
        gmm::mult(mOM, vFU, gmm::scaled(vOTU, -1),
            vFUt);
        gmm::mult(gmm::transposed(mNS), vFUt, vT);

        P.build_with(mA, 50, 1E-9);
        iter.init();
        iter.set_rhsnorm(residual);
        iter.set_noisy(1);
        iter.set_maxiter(40000);
        gmm::gmres(mA, vU, vT, P, 50, iter);
        converge = iter.converged();

        gmm::clean(vU, 1.0E-12);
        gmm::mult(mNS, vU, vTU, vFU);
        gmm::clean(vFU, 1.0E-12);
        expn.write_point_data(*mfu, vFU);
        expn.serie_add_object("option_price");

    }

    return converge;
}
```

Option B

```
#include <iostream>
#include <string>
#include <math.h>
#include "pde_fem.h"
#include "param.h"
#include <getfem/bgeot_poly_composite.h>
#include <getfem/bgeot_comma_init.h>
#include <getfem/getfem_regular_meshes.h>
#include <getfem/getfem_derivatives.h>
#include <getfem/getfem_superlu.h>
#include <getfem/getfem_export.h>
#include "analytic.h"
```

```
//#define HEAT_EQUATION 1

using namespace std;

#ifndef HEAT_EQUATION
double coeA(const base_node &x){
        return (SIGMA*SIGMA-RISK_FREE_RATE+DIVIDEND)*x[0];
}

double coeB(const base_node &x){
        return (0.5*SIGMA*SIGMA*x[0]*x[0]);
}
#endif

double init_con(const base_node &x){
#ifndef HEAT_EQUATION
        return ((x[0]>STRIKE_PRICE)? x[0]-STRIKE_PRICE:0 );
#endif
#ifdef HEAT_EQUATION
        if(x[0] >= -3.0 && x[0] < -1.0)
                return 0;
        if(x[0] >= -1.0 && x[0] <= 1.0)
                return 1.0;
        if(x[0] > 1.0 &&  x[0] <= 3.1)
                return 0;
#endif
}

double boundary_con_vanilla(const base_node &x){
        return ((x[0]> B_LEVEL)? x[0]:0);
}


PDESolve::PDESolve(){
        dim = 1;
        c = B_LEVEL;
        k = STRIKE_PRICE;
        s = CURRENT_PRICE;
        t = MATURITY;
        btype = DOWN_OUT;
        d = DIVIDEND;
        r = RISK_FREE_RATE;
        s_step = 0.1;
        t_step = 0.025;
```

```
        residual = 1E-10;
        it_number = 0;
        mymesh = new getfem::mesh;
        mfu = new getfem::mesh_fem;
        mf_coe = new getfem::mesh_fem;
        mim = new getfem::mesh_im;
}

PDESolve::~PDESolve(){
        delete mymesh;
        delete mfu;
        delete mf_coe;
        delete mim;
}

void PDESolve::setBarrierType(int b_type){
        btype = b_type;
};

void PDESolve::setVolatility(double volatility){
        sigma = volatility;
};

void PDESolve::setRiskFreeRate(double rate){
        r = rate;
};

void PDESolve::setStrike(double strike){
        k = strike;
};

void PDESolve::setBarrierLevel(double b_level){
        c = b_level;
};

void PDESolve::setMaturity(double maturity){
        t = maturity;
};

void PDESolve::setCurrentPrice(double price){
        s = price;
};

void PDESolve::setDividend(double dividend){
```

```
        d = dividend;
};

void PDESolve::setSpaceStep(double step){
        s_step = step;
};


void PDESolve::BuildMesh(double start, double end){
        int i = 0, j=0;
        switch(btype){
                case DOWN_OUT:
                #ifndef HEAT_EQUATION
                if(start == 0 && end == 0)
                {
                        start_p = c;
                        it_number = 10000;
                }
                else{
                        start_p = start;
                        it_number = (end-start_p)/s_step;

                }
                #endif
                #ifdef HEAT_EQUATION
                        start_p = start;
                        it_number = (end-start_p)/s_step;
                #endif
                break;
                case UP_OUT:
                        start_p = 0;
                        end_p = c;
                break;
                case DOWN_IN:
                break;
                case UP_IN:
                break;
                default:
                break;
        }

        bgeot::base_node org(dim);
        std::vector<bgeot::base_small_vector> vect(dim);
        bgeot::base_small_vector tmp(dim);
```

```
        std::vector<int> ref(dim);
        //here is a work-around for 1-dim case 'cause the it
           considers 1 parameter as dimension but not the
           coordinate
        //while this form fits the high dimensional case
           better than the given example
        for(i=0; i<dim; i++)
        {
                org[i] = start_p;
                for(j=0; j< dim; j++)
                        tmp[i] = s_step;
                vect[i] = tmp;
                ref[i]= it_number;
        }

        mymesh->clear();
        getfem::parallelepiped_regular_simplex_mesh(*mymesh,
           dim, org, vect.begin(), ref.begin());
}

void PDESolve::BuildFEM(bool init){
        mfu->init_with_mesh(*mymesh);
        mim->init_with_mesh(*mymesh);
        mf_coe->init_with_mesh(*mymesh);
        mfu->set_classical_finite_element(1);
        mf_coe->set_classical_finite_element(1);
        mim->set_integration_method(
                getfem::int_method_descriptor("
                   IM_EXACT_SIMPLEX(1)"));
}

void PDESolve::AsmGradBaseMatrix(sparse_matrix_type &M,
                const getfem::mesh_im &mim,
                const getfem::mesh_fem &mf,
                const getfem::mesh_fem &mfdata,
                const plain_vector &V){

        getfem::generic_assembly assem;
        assem.push_mi(mim);
        assem.push_mf(mf);
        assem.push_mf(mfdata);
        assem.push_data(V);
        assem.push_mat(M);
        if(mf.get_qdim() == 1)
```

```
                assem.set("a=data$1(#2);"
                        "M$1(#1,#1)+=comp(Base(#1).Grad(#1).
                            Base(#2))(:,:,j,i).a(i)");
        else
                assem.set("a=data$1(#2);"
                        "M$1(#1,#1)+=comp(vGrad(#1).vBase
                            (#1).Base(#2))(:,j,k,:,j,p).a(p)"
                            );
        assem.assembly();

}

void PDESolve::AsmGradGradMatrix(sparse_matrix_type &M, const
    getfem::mesh_im &mim,
                const getfem::mesh_fem &mf, const getfem::
                    mesh_fem &mfdata, const plain_vector &V){
        getfem::generic_assembly assem;
        assem.push_mi(mim);
        assem.push_mf(mf);
        assem.push_mf(mfdata);
        assem.push_data(V);
        assem.push_mat(M);
        if(mf.get_qdim() == 1)
                assem.set("a=data$1(#2);"
                        "M$1(#1,#1)+=comp(Grad(#1).Grad(#1).
                            Base(#2))(:,j,:,j,i).a(i)");
        else
                assem.set("a=data$1(#2);"
                        "M$1(#1,#1)+=sym(comp(vGrad(#1).
                            vGrad(#1).vBase(#2))(:,j,k,:,j,k
                            ,:,p).a(p))");
        assem.assembly();

}
void PDESolve::GetInitU(bool init){
        int nb_dof = 0, coe_nb_dof = 0;
        int nb_col = 0;
        nb_dof = mfu->nb_dof();
        coe_nb_dof = mf_coe->nb_dof();
        plain_vector vT(coe_nb_dof);
        sparse_matrix_type mH(nb_dof, nb_dof);
        plain_vector vR(nb_dof);
        plain_vector vTR(nb_dof);
```

```
        if(init)
        {
                getfem::interpolation_function(*mf_coe, vT,
                    init_con);
        }
        else
        {
                getfem::interpolation(*mfu_o, *mfu, vFU, vT,
                    0);
                vT[0] = 0;
                vT[coe_nb_dof-1] = 0;
        }
        //derive the linear system HU = R
        asm_mass_matrix(mH, *mim, *mfu);
        asm_source_term(vR, *mim, *mfu, *mf_coe, vT);
        //apply the boundary condition
        FindBoundary();
        if(init)
                nb_col = ApplyBoundary(0, 0);
        else
                nb_col = ApplyBoundary(0, 102.0);
        gmm::resize(mNS, nb_dof ,nb_col);
        gmm::mult(mH, vTU, gmm::scaled(vR, -1.0), vTR);
        gmm::resize(vU, nb_col);
        gmm::clear(vU);
        gmm::resize(vR, nb_col);

        gmm::mult(gmm::transposed(mNS),
                gmm::scaled(vTR, -1.0), vR);
        sparse_matrix_type mTH(nb_col, nb_dof);
        gmm::mult(gmm::transposed(mNS), mH, mTH);
        gmm::resize(mH, nb_col, nb_col);
        sparse_matrix_type mTNS(nb_dof, nb_col);
        gmm::copy(mNS, mTNS);
        gmm::mult(mTH, mTNS, mH);

        //solving the system to get U at time 0
        gmm::iteration iter(residual, 1, 40000);
        gmm::ilut_precond<sparse_matrix_type> P(mH, 50, 1E-9);
        gmm::cg(mH, vU, vR, P, iter);
}

void PDESolve::AssembleMatrix(){
        int nb_dof = 0, coe_nb_dof = 0;
```

```
int i = 0;

coe_nb_dof = mf_coe->nb_dof();
nb_dof = mfu->nb_dof();
plain_vector vA(coe_nb_dof), vB(coe_nb_dof), vT(
    coe_nb_dof);
#ifndef HEAT_EQUATION
getfem::interpolation_function(*mf_coe, vA, coeA);
getfem::interpolation_function(*mf_coe, vB, coeB);
#endif
//assembly \int_{\Omega}w^{i}w^{j}
gmm::resize(mM, nb_dof, nb_dof);
gmm::clear(mM);
asm_mass_matrix(mM, *mim, *mfu, *mf_coe);
//assembly \int_{\Omega}(\sigma^{2}-r+d)
//Sw^{i}w_{s}^{j}
gmm::resize(mA, nb_dof, nb_dof);
gmm::clear(mA);
#ifndef HEAT_EQUATION
AsmGradBaseMatrix(mA, *mim, *mfu, *mf_coe, vA);
#endif
//assembly \int_{\Omega} (1/2 \sigma^{2}S^{2}w^{j}_{s}
    w^{i}_{s})
gmm::resize(mS, nb_dof, nb_dof);
gmm::clear(mS);
//AsmGradGradMatrix(mS, mim, mfu, mf_coe, vB);
#ifndef HEAT_EQUATION
asm_stiffness_matrix_for_laplacian(mS, *mim, *mfu,
                *mf_coe, vB);
#endif
#ifdef HEAT_EQUATION
asm_stiffness_matrix_for_homogeneous_laplacian(mS,
                *mim, *mfu);
#endif

}

void PDESolve::FindBoundary(){

        getfem::mesh_region border_faces;
        getfem::outer_faces_of_mesh(*mymesh, border_faces);

        for (getfem::mr_visitor i(border_faces);
                !i.finished(); ++i) {
```

```
                assert(i.is_face());
                base_node un = mymesh->
                   normal_of_face_of_convex(i.cv(), i.f());
                un /= gmm::vect_norm2(un);
                mymesh->region(
                        DIRICHLET_BOUNDARY).add(i.cv(), i.f())
                           ;
        }
}

int PDESolve::ApplyBoundary(double time, double up){


        int nb_dof = 0, coe_nb_dof = 0;
        nb_dof = mfu->nb_dof();
        coe_nb_dof = mf_coe->nb_dof();
        gmm::resize(vTU, nb_dof);
        gmm::clear(vTU);
        gmm::resize(mNS, nb_dof, nb_dof);
        gmm::clear(mNS);
        plain_vector vT(coe_nb_dof);
        gmm::clear(vT);
        plain_vector vR(nb_dof);
        col_sparse_matrix_type mH(nb_dof, nb_dof);
        plain_vector vK(coe_nb_dof);
        gmm::clear(vK);

        #ifndef HEAT_EQUATION
        if(up == 0.0)//up boundary = 0 means it is infinite
        {
                getfem::interpolation_function(*mf_coe, vT,
                   boundary_con_vanilla);
                for(int j=0; j< coe_nb_dof; j++)
                {
                        vK[j] = -1*k*exp(-1*r*time);
                }
                gmm::add(vK, vT);
                vT[0] = 0;
        }
        else{
                gmm::clear(vT);
        }
        #endif
        getfem::asm_dirichlet_constraints(mH, vR, *mim, *mfu,
```

```
            *mf_coe, *mf_coe, vT, DIRICHLET_BOUNDARY);
        gmm::clean(mH, 1.0E-12);
        return getfem::Dirichlet_nullspace(mH, mNS, vR, vTU);

}

double PDESolve::solve(){
        double o_price = 0.0;
        int nb_dof = 0, nb_col = 0;
        int i = 0, converge=0;
        double t_time = 0.0;
        int period = 2;

        char s[100];
        gmm::iteration iter(residual, 1, 40000);
        double time = dal::uclock_sec();

        BuildMesh(0, 0);
        BuildFEM(true);
        AssembleMatrix();

        nb_dof = mfu->nb_dof();


        //get u at time 0 from initial condition
        GetInitU(true);

        nb_col = gmm::vect_size(vU);
        gmm::resize(vFU, nb_dof);
        gmm::clear(vFU);

        gmm::clean(vU, 1.0E-12);
        gmm::mult(mNS, vU, vTU, vFU);
        gmm::clean(vFU, 1.0E-12);

        //export the result as dx
        getfem::dx_export exp("solution.dx", true);
        exp.exporting(*mfu);

        exp.write_point_data(*mfu, vFU);
        exp.serie_add_object("option_price");

        //derive the linear system for time 0
        //gmm::scale(mA, -1);
```

```
        gmm::add(mS, mA);
        #ifndef HEAT_EQUATION
        gmm::add(gmm::scaled(mM,r), mA);
        #endif
        gmm::scale(mM, 1/t_step);
        gmm::add(mM, mA);

        //store the final result with boundary condition
        plain_vector vFUt(nb_dof);
        gmm::clear(vFUt);

        plain_vector vT(nb_col);
        gmm::clear(vT);

        plain_vector vOTU(nb_dof);
        gmm::clear(vOTU);

        sparse_matrix_type mTA(nb_col, nb_dof);
        sparse_matrix_type mOA(nb_dof, nb_dof);
        sparse_matrix_type mOM(nb_dof, nb_dof);
        sparse_matrix_type mTM(nb_col, nb_dof);
        sparse_matrix_type mTNS(nb_dof, nb_col);
        gmm::copy(mA, mOA);
        gmm::copy(mM, mOM);
        gmm::resize(mA, nb_col, nb_col);
        gmm::resize(mM, nb_col, nb_col);

        gmm::ilut_precond<sparse_matrix_type> P;

        for(i=0; i< (t-0.2)/t_step; i++){

                t_time = t_step*(i+1);
                ApplyBoundary(t_time, 0);
                gmm::resize(mNS, nb_dof ,nb_col);

                gmm::mult(gmm::transposed(mNS), mOA, mTA);
                gmm::mult(gmm::transposed(mNS), mOM, mTM);
                gmm::copy(mNS, mTNS);
                gmm::mult(mTA, mTNS, mA);
                gmm::mult(mTM, mTNS, mM);
                gmm::mult(mOA, vTU, vOTU);
                gmm::mult(mOM, vFU, gmm::scaled(vOTU, -1),
                   vFUt);
                gmm::mult(gmm::transposed(mNS), vFUt, vT);
```

```
P.build_with(mA, 50, 1E-9);
iter.init();
iter.set_rhsnorm(residual);
iter.set_noisy(1);
iter.set_maxiter(40000);
//gmm::cg(mA, vU, vT, P,  iter);
gmm::gmres(mA, vU, vT, P, 50, iter);
converge = iter.converged();

gmm::clean(vU, 1.0E-12);
gmm::mult(mNS, vU, vTU, vFU);
gmm::clean(vFU, 1.0E-12);
exp.write_point_data(*mfu, vFU);
exp.serie_add_object("option_price");

}

//next period
t_step = 0.01;
s_step = 0.05;
mymesh_o = mymesh;
mymesh = new getfem::mesh;
BuildMesh(80.0, 102);
mfu_o = mfu;
mf_coe_o = mf_coe;
mf_coe = new getfem::mesh_fem;
mfu = new getfem::mesh_fem;
mim_o = mim;
mim = new getfem::mesh_im;
BuildFEM(false);
AssembleMatrix();
//get u at time t+1 from time t

nb_dof = mfu->nb_dof();
GetInitU(false);
mymesh_o->write_to_file("bs_mesh_o.msh");
delete mymesh_o;
delete mfu_o;
delete mf_coe_o;
delete mim_o;
mymesh->write_to_file("bs_mesh.msh");

gmm::add(mS, mA);
```

```
#ifndef HEAT_EQUATION
gmm::add(gmm::scaled(mM,r), mA);
#endif
gmm::scale(mM, 1/t_step);
gmm::add(mM, mA);

nb_col = gmm::vect_size(vU);
gmm::resize(mTA, nb_col, nb_dof);
gmm::resize(mOA, nb_dof, nb_dof);
gmm::resize(mOM, nb_dof, nb_dof);
gmm::resize(mTM, nb_col, nb_dof);
gmm::copy(mA, mOA);
gmm::copy(mM, mOM);
gmm::resize(mA, nb_col, nb_col);
gmm::resize(mM, nb_col, nb_col);


gmm::resize (mTNS, nb_dof, nb_col);


getfem::dx_export expn("solution_1.dx", true);
expn.exporting(*mfu);


gmm::ilut_precond<sparse_matrix_type> PN(mA,
                                         50, 1E-9);

//store the final result with boundary condition
gmm::resize(vFUt, nb_dof);
gmm::resize(vFU, nb_dof);
gmm::clear(vFU);
gmm::clear(vFUt);

gmm::resize(vOTU ,nb_dof);
gmm::clear(vOTU);

gmm::resize (vT, nb_col);
gmm::clear(vT);

gmm::clean(vU, 1.0E-12);
gmm::mult(mNS, vU, vTU, vFU);
gmm::clean(vFU, 1.0E-12);
expn.write_point_data(*mfu, vFU);
expn.serie_add_object("option_price");
```

```
for(i=0; i< 0.2/t_step; i++){

        t_time = t_step*(i+1);
        ApplyBoundary(t_time, 102.0);
        gmm::resize(mNS, nb_dof ,nb_col);

        gmm::mult(gmm::transposed(mNS), mOA, mTA);
        gmm::mult(gmm::transposed(mNS), mOM, mTM);
        gmm::copy(mNS, mTNS);
        gmm::mult(mTA, mTNS, mA);
        gmm::mult(mTM, mTNS, mM);

        gmm::mult(mOA, vTU, vOTU);
        gmm::mult(mOM, vFU, gmm::scaled(vOTU, -1),
           vFUt);
        gmm::mult(gmm::transposed(mNS), vFUt, vT);

        P.build_with(mA, 50, 1E-9);
        iter.init();
        iter.set_rhsnorm(residual);
        iter.set_noisy(1);
        iter.set_maxiter(40000);
        //gmm::cg(mA, vU, vT, P,  iter);
        gmm::gmres(mA, vU, vT, P, 50, iter);
        converge = iter.converged();

        gmm::clean(vU, 1.0E-12);
        gmm::mult(mNS, vU, vTU, vFU);
        gmm::clean(vFU, 1.0E-12);
        expn.write_point_data(*mfu, vFU);
        expn.serie_add_object("option_price");

    }

    return converge;
}
```

B.2 Heston model

B.2.1 Vanilla options

```
#include <iostream>
#include <string>
```

```
#include <math.h>
#include "heston_pde_fem.h"
#include "param.h"
#include <getfem/bgeot_poly_composite.h>
#include <getfem/bgeot_comma_init.h>
#include <getfem/getfem_regular_meshes.h>
#include <getfem/getfem_derivatives.h>
#include <getfem/getfem_superlu.h>
#include <getfem/getfem_export.h>
#include <getfem/getfem_import.h>


using namespace std;

double coe_mu1(const base_node &x){
        return (RD-RF-0.5*x[1]-0.5*XI*RHO);
}

double coe_mu2(const base_node &x){
        return (KAPPA*(THETA - x[1]) - 0.5*XI*XI);
}

double coe_sigma11(const base_node &x){
        return (0.5*x[1]);
}

double coe_sigma12(const base_node &x){
        return (0.5*x[1]*XI*RHO);
}

double coe_sigma22(const base_node &x){
        return (0.5*x[1]*XI*XI);
}

double init_con_st(const base_node &x){
        return ((exp(x[0])>STRIKE)? exp(x[0])-STRIKE:0 );
}

double neumann_boundary_con(const base_node &x){
        return (0.5*x[1]*exp(x[0]));
}

HPDESolve::HPDESolve(){
        dim = 2;
```

```
        cd = DOWN_B;
        cu = 9;
        k = log(100);
        t = 0.5;
        rd = RD;
        rf = RF;
        s_step = 0.1;
        t_step = 0.025;
        residual = 1E-10;
        it_number = 0;
        mymesh = new getfem::mesh;
        mfu = new getfem::mesh_fem;
        mf_coe = new getfem::mesh_fem;
        mim = new getfem::mesh_im;
        mesh_file = "gmsh:stv.msh";
}

HPDESolve::~HPDESolve(){
        delete mymesh;
        delete mfu;
        delete mf_coe;
        delete mim;
}

void HPDESolve::BuildMesh(double down, double up){
        getfem::import_mesh(mesh_file, *mymesh);
}

void HPDESolve::FindBoundary(){

        getfem::mesh_region border_faces;
        getfem::outer_faces_of_mesh(*mymesh, border_faces);

for (getfem::mr_visitor i(border_faces); !i.finished()
    ; ++i) {
        assert(i.is_face());
        base_node un = mymesh->
            normal_of_face_of_convex(i.cv(), i.f());
        un /= gmm::vect_norm2(un);
        if(un[0] < 0)//dirichlet boundary condition
            when $e^{x} = c_{d}$
        {
                mymesh->region(DIRICHLET_BOUNDARY_X).
                    add(i.cv(), i.f());
```

```
                }
                else if(un[dim-2] > 0)//neumann boundary
                    condition when $e^{x} = \infty$
                {
                        mymesh->region(NEUMANN_BOUNDARY_X).add
                            (i.cv(), i.f());
                }else if(un[dim-1] < 0)
                {

                        mymesh->region(DIRICHLET_BOUNDARY_V).
                            add(i.cv(), i.f());
                }
                else if(gmm::abs(un[dim-1]))//neumann boundary
                    condition when $v=0, \infty$
                {
                        mymesh->region(NEUMANN_BOUNDARY_V).add
                            (i.cv(), i.f());
                }
        }

}



int HPDESolve::ApplyBoundary(double time, double up){


        int nb_dof = 0, coe_nb_dof = 0;
        int i = 0;
        nb_dof = mfu->nb_dof();
        coe_nb_dof = mf_coe->nb_dof();
                gmm::resize(vTU, nb_dof);
        gmm::clear(vTU);
        gmm::resize(mNS, nb_dof, nb_dof);
        gmm::clear(mNS);
        plain_vector vT(coe_nb_dof);
        gmm::clear(vT);
        plain_vector vR(nb_dof);
        col_sparse_matrix_type mH(nb_dof, nb_dof);
        plain_vector vK(coe_nb_dof);
        gmm::clear(vK);

        sparse_matrix_type mTS(nb_dof, nb_dof);
```

```
if(up == 0.0)//up boundary = 0 means it is infinite
{
        //neumman boundary condition
        getfem::interpolation_function(*mf_coe, vT,
           neumann_boundary_con);
        for(int j=0; j< coe_nb_dof; j++)
        {
                vK[j] = vT[j]*exp(-1*rf*time) ;
        }

        gmm::resize(vS, coe_nb_dof);
        gmm::clear(vS);
        asm_source_term(vS, *mim, *mfu, *mf_coe, vK,
           NEUMANN_BOUNDARY_X);
        gmm::clear(vK);
        asm_source_term(vS, *mim, *mfu, *mf_coe, vK,
           NEUMANN_BOUNDARY_V);
        gmm::clear(vT);

    }
    else{
        gmm::clear(vT);
    }
    getfem::asm_dirichlet_constraints(mH, vR, *mim, *mfu,
       *mf_coe, *mf_coe, vT, DIRICHLET_BOUNDARY_X);
    gmm::clean(mH, 1.0E-12);
    return getfem::Dirichlet_nullspace(mH, mNS, vR, vTU);

}

void HPDESolve::BuildFEM(bool init){
    int qdim = 0;
    mfu->init_with_mesh(*mymesh);
    mim->init_with_mesh(*mymesh);
    mf_coe->init_with_mesh(*mymesh);
    mfu->set_classical_finite_element(1);
    mf_coe->set_classical_finite_element(1);
    mim->set_integration_method(getfem::
       int_method_descriptor("IM_EXACT_SIMPLEX(2)"));
    qdim = mfu->get_qdim();

}
```

```
void HPDESolve::AsmGradBaseMatrix(sparse_matrix_type &M,
                const getfem::mesh_im &mim,
                const getfem::mesh_fem &mf,
                const getfem::mesh_fem &mfdata,
                const plain_vector &V,
                const getfem::mesh_region
                &rg = getfem::mesh_region::all_convexes()){

        getfem::generic_assembly assem;
        assem.push_mi(mim);
        assem.push_mf(mf);
        assem.push_mf(mfdata);
        assem.push_data(V);
        assem.push_mat(M);
        if(mf.get_qdim() == 1)
                assem.set("a=data$1(mdim(#1), #2);"
                        "M$1(#1,#1)+=comp(Base(#1).Grad(#1).
                            Base(#2))(:,:,j,i).a(j, i)");
        assem.assembly(rg);

}

void HPDESolve::AssembleMatrix(){
        int nb_dof = 0, coe_nb_dof = 0;
        int i = 0;

        coe_nb_dof = mf_coe->nb_dof();
        nb_dof = mfu->nb_dof();
        plain_vector vA(coe_nb_dof), vB(coe_nb_dof), vT(
            coe_nb_dof);
        plain_vector vSI11(coe_nb_dof), vSI12(coe_nb_dof),
            vSI22(coe_nb_dof);
        plain_vector vMU(2*coe_nb_dof);
        plain_vector vSI(4*coe_nb_dof);

        getfem::interpolation_function(*mf_coe, vA, coe_mu1);
        getfem::interpolation_function(*mf_coe, vB, coe_mu2);
        for(i =0; i< coe_nb_dof; i++){
                vMU[2*i] = vA[i];
                vMU[2*i+1] = vB[i];
        }
        gmm::resize(mA, nb_dof, nb_dof);
        gmm::clear(mA);
```

```
AsmGradBaseMatrix(mA, *mim, *mfu, *mf_coe, vMU);
getfem::interpolation_function(*mf_coe, vSI11,
   coe_sigma11);
getfem::interpolation_function(*mf_coe, vSI12,
   coe_sigma12);
getfem::interpolation_function(*mf_coe, vSI22,
   coe_sigma22);
for(i = 0; i<coe_nb_dof; i++){
        vSI[4*i] = vSI11[i];
        vSI[4*i+1] = vSI12[i];
        vSI[4*i+2] = vSI12[i];
        vSI[4*i+3] = vSI22[i];
      }

      gmm::resize(mS, nb_dof, nb_dof);
      gmm::clear(mS);
      asm_stiffness_matrix_for_scalar_elliptic(mS, *mim,
                    *mfu, *mf_coe, vSI);
      gmm::resize(mM, nb_dof, nb_dof);
      gmm::clear(mM);
      asm_mass_matrix(mM, *mim, *mfu, *mf_coe);

}

void HPDESolve::MinVolBoundary(double time){

      int nb_dof = 0;
      int i = 0;
      double r = 0;
      nb_dof = mfu->nb_dof();
      plain_vector vK(nb_dof);
      gmm::clear(vK);
      plain_vector vMU(2*nb_dof);
      gmm::clear(vMU);
      plain_vector vVU(nb_dof);

      sparse_matrix_type mTA;
      sparse_matrix_type mTM;

      r = rd-rf-0.5*XI*RHO;
      for(i=0; i<2*nb_dof; i++){
            vMU[2*i] = r;
      }
```

```
gmm::resize(mTA, nb_dof, nb_dof);
gmm::resize(mTM, nb_dof, nb_dof);
//linear system mTA*u^{n+1} = mTM*u^{n}
asm_mass_matrix(mTM, *mim, *mfu, *mf_coe,
   DIRICHLET_BOUNDARY_V);
AsmGradBaseMatrix(mTA, *mim, *mfu, *mf_coe, vMU,
   DIRICHLET_BOUNDARY_V);

gmm::scale(mTA, -1);
gmm::add(gmm::scaled(mTM, rd), mTA);
gmm::scale(mTM, 1/t_step);
gmm::add(mTM, mTA);
//boundary condition s->0;s->\infty
getfem::interpolation_function(*mf_coe, vK,
   init_con_st);

sparse_matrix_type mH(nb_dof, nb_dof);
plain_vector vR(nb_dof);

//derive the linear system HU = R for the init U
asm_mass_matrix(mH, *mim, *mfu, DIRICHLET_BOUNDARY_V);
asm_source_term(vR, *mim, *mfu, *mf_coe, vK,
   DIRICHLET_BOUNDARY_V);
gmm::clean(mH, 1.0E-12);
//solving the system to get U at time 0
gmm::iteration iter(residual, 1, 40000);
gmm::identity_matrix P;
iter.init();
iter.set_rhsnorm(residual);
iter.set_noisy(1);
iter.set_maxiter(40000);

gmm::resize(vVU, nb_dof);
gmm::clear(vVU);

gmm::cg(mH, vVU, vR, P, iter);
gmm::clean(vVU, 1.0E-12);

iter.init();
iter.set_rhsnorm(residual);
iter.set_noisy(1);
iter.set_maxiter(40000);
```

```
        gmm::clear(vK);
        gmm::mult(mTM, vVU, vK);
        gmm::gmres(mTA, vVU, vK, P, 50, iter);
        gmm::clean(vVU, 1.0E-12);

}

void HPDESolve::GetInitU(bool init){
        int nb_dof = 0, coe_nb_dof = 0;
        int nb_col = 0;
        nb_dof = mfu->nb_dof();
        coe_nb_dof = mf_coe->nb_dof();
        plain_vector vT(coe_nb_dof);
        sparse_matrix_type mH(nb_dof, nb_dof);
        plain_vector vR(nb_dof);
        plain_vector vTR(nb_dof);
        gmm::resize(vFU, nb_dof);
        gmm::clear(vFU);

        if(init)
        {
                getfem::interpolation_function(*mf_coe, vT,
                    init_con_st);
        }
        else
        {
                getfem::interpolation(*mfu_o, *mfu, vFU, vT,
                    0);
        }
        //derive the linear system HU = R
        asm_mass_matrix(mH, *mim, *mfu);
        asm_source_term(vR, *mim, *mfu, *mf_coe, vT);
        //apply the boundary condition
        nb_col = ApplyBoundary(0, 0);
        gmm::resize(mNS, nb_dof ,nb_col);
        gmm::mult(mH, vTU, gmm::scaled(vR, -1.0), vTR);
        gmm::resize(vU, nb_col);
        gmm::clear(vU);
        gmm::resize(vR, nb_col);

        gmm::mult(gmm::transposed(mNS), gmm::scaled(vTR, -1.0)
            , vR);
        sparse_matrix_type mTH(nb_col, nb_dof);
```

```
        gmm::mult(gmm::transposed(mNS), mH, mTH);
        gmm::resize(mH, nb_col, nb_col);
        sparse_matrix_type mTNS(nb_dof, nb_col);
        gmm::copy(mNS, mTNS);
        gmm::mult(mTH, mTNS, mH);

        //solving the system to get U at time 0
        gmm::iteration iter(residual, 1, 40000);
        gmm::ilut_precond<sparse_matrix_type> P(mH, 50, 1E-9);
        gmm::cg(mH, vU, vR, P, iter);

        gmm::clean(vU, 1.0E-12);
        gmm::mult(mNS, vU, vTU, vFU);
        gmm::clean(vFU, 1.0E-12);

}

double HPDESolve::solve(){
        double o_price = 0.0;
        int nb_dof = 0, nb_col = 0;
        int i = 0;
        double t_time = 0.0;
        int period = 2;

        char s[100];
        gmm::iteration iter(residual, 1, 40000);
        double time = dal::uclock_sec();

        double converge = 0;
        BuildMesh(0, 0);
        FindBoundary();
        BuildFEM(true);
        mymesh->write_to_file("stv_mesh.msh");

        //MinVolBoundary(0);
        AssembleMatrix();


        nb_dof = mfu->nb_dof();
        //derive the linear system for time 0
        gmm::scale(mA, -1);
        gmm::add(mS, mA);
        gmm::add(gmm::scaled(mM,rd), mA);
        gmm::scale(mM, 1/t_step);
```

```
gmm::add(mM, mA);

//apply the boundary and get additional source term
GetInitU(true);

nb_col = gmm::vect_size(vU);
//export the result as dx
getfem::dx_export exp("solution.dx", true);
exp.exporting(*mfu);

exp.write_point_data(*mfu, vFU);
exp.serie_add_object("option_price");

plain_vector vFUt(nb_dof);
gmm::clear(vFUt);

plain_vector vT(nb_col);
gmm::clear(vT);

plain_vector vOTU(nb_dof);
gmm::clear(vOTU);

sparse_matrix_type mTA(nb_col, nb_dof);
sparse_matrix_type mOA(nb_dof, nb_dof);
sparse_matrix_type mOM(nb_dof, nb_dof);
sparse_matrix_type mTM(nb_col, nb_dof);
sparse_matrix_type mTNS(nb_dof, nb_col);
gmm::copy(mA, mOA);
gmm::copy(mM, mOM);
gmm::resize(mA, nb_col, nb_col);
gmm::resize(mM, nb_col, nb_col);

gmm::ilut_precond<sparse_matrix_type> P;

for(i=1; i< t/t_step; i++){

        t_time = t_step*i;
        nb_col = ApplyBoundary(t_time, 0);
        gmm::resize(mNS, nb_dof ,nb_col);

        gmm::mult(gmm::transposed(mNS), mOA, mTA);
        gmm::mult(gmm::transposed(mNS), mOM, mTM);
        gmm::copy(mNS, mTNS);
        gmm::mult(mTA, mTNS, mA);
```

```
            gmm::mult(mTM, mTNS, mM);
            gmm::mult(mOA, vTU, vOTU);
            gmm::mult(mOM, vFU, gmm::scaled(vOTU, -1),
               vFUt);
            gmm::add(vS, vFUt);
            gmm::mult(gmm::transposed(mNS), vFUt, vT);

            P.build_with(mA, 50, 1E-9);
            iter.init();
            iter.set_rhsnorm(residual);
            iter.set_noisy(1);
            iter.set_maxiter(40000);
            gmm::gmres(mA, vU, vT, P, 50, iter);
            converge = iter.converged();

            gmm::clean(vU, 1.0E-12);
            gmm::mult(mNS, vU, vTU, vFU);
            gmm::clean(vFU, 1.0E-12);
            exp.write_point_data(*mfu, vFU);
            exp.serie_add_object("option_price");

    }

        return converge;
}
```

B.2.2 Multi-windowed barrier options

```
#include <iostream>
#include <string>
#include <math.h>
#include "heston_pde_fem.h"
#include "param.h"
#include <getfem/bgeot_poly_composite.h>
#include <getfem/bgeot_comma_init.h>
#include <getfem/getfem_regular_meshes.h>
#include <getfem/getfem_derivatives.h>
#include <getfem/getfem_superlu.h>
#include <getfem/getfem_export.h>
#include <getfem/getfem_import.h>
#include <getfem/getfem_mesh_slice.h>


using namespace std;
```

```
double coe_mu1(const base_node &x){
        return (RD-RF-0.5*x[1]-0.5*XI*RHO);
}

double coe_mu2(const base_node &x){
        return (KAPPA*(THETA - x[1]) - 0.5*XI*XI);
}

double coe_sigma11(const base_node &x){
        return (0.5*x[1]);
}

double coe_sigma12(const base_node &x){
        return (0.5*x[1]*XI*RHO);
}

double coe_sigma22(const base_node &x){
        return (0.5*x[1]*XI*XI);
}

double init_con_st(const base_node &x){
        if(exp(x[0])> STRIKE && x[0] < UP_B)
                return (exp(x[0]) - STRIKE);
        else return 0;
}

double neumann_boundary_con(const base_node &x){
        return (0.5*x[1]*exp(x[0]));
}

HPDESolve::HPDESolve(){
        dim = 2;
        cd = DOWN_B;
        cu = UP_B;
        k = log(100);
        t = 0.5;
        rd = RD;
        rf = RF;
        s_step = 0.1;
        t_step = 0.005;
        residual = 1E-10;
        it_number = 0;
        mymesh = new getfem::mesh;
        mfu = new getfem::mesh_fem;
```

```
        mf_coe = new getfem::mesh_fem;
        mim = new getfem::mesh_im;
        mesh_file = "gmsh:stv.msh";
        mesh_file_n = "gmsh:stv_n.msh";
}

HPDESolve::~HPDESolve(){
        delete mymesh;
        delete mfu;
        delete mf_coe;
        delete mim;
}

void HPDESolve::BuildMesh(bool init){
        if(init)
                getfem::import_mesh(mesh_file, *mymesh);
        else getfem::import_mesh(mesh_file_n, *mymesh);
}

void HPDESolve::FindBoundary(double down, double up){

        getfem::mesh_region border_faces;
        getfem::outer_faces_of_mesh(*mymesh, border_faces);

        for (getfem::mr_visitor i(border_faces); !i.finished()
            ; ++i) {
                assert(i.is_face());
                base_node un = mymesh->
                    normal_of_face_of_convex(i.cv(), i.f());
                un /= gmm::vect_norm2(un);
                if(un[0] < 0)//dirichlet boundary condition
                    when $e^{x} = c_{d}$
                {
                        mymesh->region(DIRICHLET_BOUNDARY_X).
                            add(i.cv(), i.f());
                        mymesh->region(DIRICHLET_BOUNDARY_XV).
                            add(i.cv(), i.f());
                }
                else if(un[dim-2] > 0)//neumann boundary
                    condition when $e^{x} = \infty$
                {
                        if(up == 0){ //means infinity
                                mymesh->region(
                                    NEUMANN_BOUNDARY_XV).add(i.
```

```
                    cv(), i.f());
                mymesh->region(
                    NEUMANN_BOUNDARY_X).add(i.
                    cv(), i.f());
        }else
        {
                mymesh->region(
                    DIRICHLET_BOUNDARY_XV).add(
                    i.cv(), i.f());
                mymesh->region(
                    DIRICHLET_BOUNDARY_X).add(i
                    .cv(), i.f());

                }

        }else if(un[dim-1] < 0)
        {
                mymesh->region(DIRICHLET_BOUNDARY_V).
                    add(i.cv(), i.f());
                mymesh->region(DIRICHLET_BOUNDARY_XV).
                    add(i.cv(), i.f());
                mymesh->region(NEUMANN_BOUNDARY_V).add
                    (i.cv(), i.f());
        }
        else if(un[dim-1] > 0)//neumann boundary
            condition when $v=0, \infty$
        {
                mymesh->region(NEUMANN_BOUNDARY_XV).
                    add(i.cv(), i.f());
                mymesh->region(NEUMANN_BOUNDARY_V).add
                    (i.cv(), i.f());
        }
        }
}

int HPDESolve::ApplyBoundary(double time, double up){


        int nb_dof = 0, coe_nb_dof = 0;
        int i = 0;
        nb_dof = mfu->nb_dof();
        coe_nb_dof = mf_coe->nb_dof();
        gmm::resize(vTU, nb_dof);
```

```
        gmm::clear(vTU);
        gmm::resize(mNS, nb_dof, nb_dof);
        gmm::clear(mNS);
        gmm::resize(vS, nb_dof);
        gmm::clear(vS);

        plain_vector vT(coe_nb_dof);
        gmm::clear(vT);
        plain_vector vR(nb_dof);
        col_sparse_matrix_type mH(nb_dof, nb_dof);
        plain_vector vK(2*coe_nb_dof);
        gmm::clear(vK);

        if(up == 0.0)//up boundary = 0 means it is infinite
        {
                //neumman boundary condition
                getfem::interpolation_function(*mf_coe, vT,
                   neumann_boundary_con);
                for(int j=0; j< coe_nb_dof; j++)
                {
                        vK[2*j] = vT[j]*exp(-1*rf*time) ;
                }
                asm_normal_source_term(vS, *mim, *mfu,
                                *mf_coe, vK,
                                   NEUMANN_BOUNDARY_X);
        }
        gmm::clear(vT);
        asm_source_term(vS, *mim, *mfu, *mf_coe, vT,
           NEUMANN_BOUNDARY_V);
        gmm::clear(vT);
        getfem::asm_dirichlet_constraints(mH, vR, *mim, *mfu,
           *mf_coe, *mf_coe, vT, DIRICHLET_BOUNDARY_X);
        gmm::clean(mH, 1.0E-12);
        return getfem::Dirichlet_nullspace(mH, mNS, vR, vTU);

}

void HPDESolve::BuildFEM(bool init){
        int qdim = 0;
        mfu->init_with_mesh(*mymesh);
        mim->init_with_mesh(*mymesh);
        mf_coe->init_with_mesh(*mymesh);
        mfu->set_finite_element(
                        getfem::fem_descriptor("FEM_PK(2, 1)")
```

```
                                    );
        mf_coe->set_finite_element(
                            getfem::fem_descriptor("FEM_PK(2, 1)")
                                    );
        mim->set_integration_method(
                getfem::int_method_descriptor("IM_TRIANGLE(6)"
                    ));
        qdim = mfu->get_qdim();

}

void HPDESolve::AsmGradBaseMatrix(sparse_matrix_type &M,
                const getfem::mesh_im &mim,
                const getfem::mesh_fem &mf,
                const getfem::mesh_fem &mfdata,
                const plain_vector &V,
                const getfem::mesh_region
                &rg = getfem::mesh_region::all_convexes()){

        getfem::generic_assembly assem;
        assem.push_mi(mim);
        assem.push_mf(mf);
        assem.push_mf(mfdata);
        assem.push_data(V);
        assem.push_mat(M);
        if(mf.get_qdim() == 1)
                assem.set("a=data$1(mdim(#1), #2);"
                        "M$1(#1,#1)+=comp(Base(#1).Grad(#1).
                            Base(#2))(:,:,j,i).a(j, i)");
        assem.assembly(rg);

}

void HPDESolve::AssembleMatrix(){
        int nb_dof = 0, coe_nb_dof = 0;
        int i = 0;

        coe_nb_dof = mf_coe->nb_dof();
        nb_dof = mfu->nb_dof();
        plain_vector vA(coe_nb_dof), vB(coe_nb_dof),
                        vT(coe_nb_dof);
        plain_vector vSI11(coe_nb_dof), vSI12(coe_nb_dof),
                        vSI22(coe_nb_dof);
        plain_vector vMU(2*coe_nb_dof);
```

```
        plain_vector vSI(4*coe_nb_dof);

        getfem::interpolation_function(*mf_coe, vA, coe_mu1);
        getfem::interpolation_function(*mf_coe, vB, coe_mu2);
        for(i =0; i< coe_nb_dof; i++){
                vMU[2*i] = vA[i];
                vMU[2*i+1] = vB[i];
        }
        gmm::resize(mA, nb_dof, nb_dof);
        gmm::clear(mA);

        AsmGradBaseMatrix(mA, *mim, *mfu, *mf_coe, vMU);

        getfem::interpolation_function(*mf_coe, vSI11,
           coe_sigma11);
        getfem::interpolation_function(*mf_coe, vSI12,
           coe_sigma12);
        getfem::interpolation_function(*mf_coe, vSI22,
           coe_sigma22);
        for(i = 0; i<coe_nb_dof; i++){
                vSI[4*i] = vSI11[i];
                vSI[4*i+1] = vSI12[i];
                vSI[4*i+2] = vSI12[i];
                vSI[4*i+3] = vSI22[i];
        }

        gmm::resize(mS, nb_dof, nb_dof);
        gmm::clear(mS);
        asm_stiffness_matrix_for_scalar_elliptic(mS, *mim,
                        *mfu, *mf_coe, vSI);
        gmm::resize(mM, nb_dof, nb_dof);
        gmm::clear(mM);
        asm_mass_matrix(mM, *mim, *mfu, *mf_coe);

}

void HPDESolve::MinVolBoundary(double time){

        int nb_dof = 0;
        int i = 0;
        double r = 0;
        nb_dof = mfu->nb_dof();
        plain_vector vK(nb_dof);
        gmm::clear(vK);
```

```
plain_vector vMU(2*nb_dof);
gmm::clear(vMU);
gmm::resize(vVU, nb_dof);
gmm::clear(vVU);

sparse_matrix_type mTA;
sparse_matrix_type mTM;
r = rd-rf;
for(i=0; i<nb_dof; i++){
        vMU[2*i] = r;
}

gmm::resize(mTA, nb_dof, nb_dof);
gmm::resize(mTM, nb_dof, nb_dof);
//linear system mTA*u^{n+1} = mTM*u^{n}
asm_mass_matrix(mTM, *mim, *mfu, *mf_coe,
   DIRICHLET_BOUNDARY_V);
AsmGradBaseMatrix(mTA, *mim, *mfu, *mf_coe, vMU,
   DIRICHLET_BOUNDARY_V);

gmm::scale(mTA, -1);
gmm::add(gmm::scaled(mTM, rd), mTA);
gmm::scale(mTM, 1/t_step);
gmm::add(mTM, mTA);
//boundary condition s->0;s->\infty
getfem::interpolation_function(*mf_coe, vK,
   init_con_st);

sparse_matrix_type mH(nb_dof, nb_dof);
plain_vector vR(nb_dof);

//derive the linear system HU = R for the init U
asm_mass_matrix(mH, *mim, *mfu, DIRICHLET_BOUNDARY_V);
asm_source_term(vR, *mim, *mfu, *mf_coe, vK,
   DIRICHLET_BOUNDARY_V);
gmm::clean(mH, 1.0E-12);
//solving the system to get U at time 0
gmm::iteration iter(residual, 1, 40000);
gmm::identity_matrix P;
iter.init();
iter.set_rhsnorm(residual);
iter.set_noisy(1);
iter.set_maxiter(40000);
```

```
        gmm::resize(vVU, nb_dof);
        gmm::clear(vVU);

        gmm::cg(mH, vVU, vR, P, iter);
        gmm::clean(vVU, 1.0E-12);

        //P.build_with(mTA, 50, 1E-9);

        for(i=1; i<time/t_step; i++)
        {
                iter.init();
                iter.set_rhsnorm(residual);
                iter.set_noisy(1);
                iter.set_maxiter(40000);

                gmm::clear(vK);
                gmm::mult(mTM, vVU, vK);
                gmm::gmres(mTA, vVU, vK, P, 50, iter);
                gmm::clean(vVU, 1.0E-12);
        }


}

void HPDESolve::GetInitU(bool init){
        int nb_dof = 0, coe_nb_dof = 0;
        int nb_col = 0;
        nb_dof = mfu->nb_dof();
        coe_nb_dof = mf_coe->nb_dof();
        plain_vector vT(coe_nb_dof);
        sparse_matrix_type mH(nb_dof, nb_dof);
        plain_vector vR(nb_dof);
        plain_vector vTR(nb_dof);

        if(init)
        {
                getfem::interpolation_function(*mf_coe, vT,
                    init_con_st);
        }
        else
        {
                getfem::interpolation(*mfu_o, *mfu, vFU, vT,
                    0);
        }
```

```
    //derive the linear system HU = R
    asm_mass_matrix(mH, *mim, *mfu);
    asm_source_term(vR, *mim, *mfu, *mf_coe, vT);
    //apply the boundary condition
    nb_col = ApplyBoundary(0, cu);
    gmm::resize(mNS, nb_dof ,nb_col);
    gmm::mult(mH, vTU,
                    gmm::scaled(vR, -1.0), vTR);
    gmm::resize(vU, nb_col);
    gmm::clear(vU);
    gmm::resize(vR, nb_col);

    gmm::mult(gmm::transposed(mNS), gmm::scaled(vTR, -1.0)
        , vR);
    sparse_matrix_type mTH(nb_col, nb_dof);
    gmm::mult(gmm::transposed(mNS), mH, mTH);
    gmm::resize(mH, nb_col, nb_col);
    sparse_matrix_type mTNS(nb_dof, nb_col);
    gmm::copy(mNS, mTNS);
    gmm::mult(mTH, mTNS, mH);

    //solving the system to get U at time 0
    gmm::iteration iter(residual, 1, 40000);
    gmm::ilut_precond<sparse_matrix_type> P(mH, 50, 1E-9);
    gmm::cg(mH, vU, vR, P, iter);
    gmm::resize(vFU, nb_dof);
    gmm::clear(vFU);


    gmm::clean(vU, 1.0E-12);
    gmm::mult(mNS, vU, vTU, vFU);
    gmm::clean(vFU, 1.0E-12);

}

double HPDESolve::solve(){
        double o_price = 0.0;
        int nb_dof = 0, nb_col = 0;
        int i = 0;
        double t_time = 0.0;
        int period = 2;

        char s[100];
        gmm::iteration iter(residual, 1, 40000);
```

```
double time = dal::uclock_sec();

double converge = 0;
BuildMesh(true);
FindBoundary(cd, cu);
BuildFEM(true);
mymesh->write_to_file("stv_mesh.msh");

//MinVolBoundary(0);
AssembleMatrix();

nb_dof = mfu->nb_dof();
//derive the linear system for time 0
gmm::scale(mA, -1);
gmm::add(mS, mA);
gmm::add(gmm::scaled(mM,rd), mA);
gmm::scale(mM, 1/t_step);
gmm::add(mM, mA);

//apply the boundary and get additional source term
GetInitU(true);

nb_col = gmm::vect_size(vU);
//export the result as dx

getfem::dx_export expsl("solution_sl.dx", true);
getfem::stored_mesh_slice sl;
sl.build(*mymesh,
        getfem::slicer_half_space(base_node(5,0.2),
                base_node(0, 1), 0), 4);

expsl.exporting(sl);
expsl.write_point_data(*mfu, vFU);
expsl.serie_add_object("option_price");

getfem::dx_export exp("solution.dx", true);
exp.exporting(*mfu);

exp.write_point_data(*mfu, vFU);
exp.serie_add_object("option_price");

plain_vector vFUt(nb_dof);
gmm::clear(vFUt);
```

```
plain_vector vT(nb_col);
gmm::clear(vT);

plain_vector vOTU(nb_dof);
gmm::clear(vOTU);

sparse_matrix_type mTA(nb_col, nb_dof);
sparse_matrix_type mOA(nb_dof, nb_dof);
sparse_matrix_type mOM(nb_dof, nb_dof);
sparse_matrix_type mTM(nb_col, nb_dof);
sparse_matrix_type mTNS(nb_dof, nb_col);
gmm::copy(mA, mOA);
gmm::copy(mM, mOM);
gmm::resize(mA, nb_col, nb_col);
gmm::resize(mM, nb_col, nb_col);

gmm::ilut_precond<sparse_matrix_type> P;

for(i=1; i<= 0.2/t_step; i++){
        t_time = t_step*i;
        nb_col = ApplyBoundary(t_time, cu);
        gmm::resize(mNS, nb_dof ,nb_col);

        gmm::mult(gmm::transposed(mNS), mOA, mTA);
        gmm::mult(gmm::transposed(mNS), mOM, mTM);
        gmm::copy(mNS, mTNS);
        gmm::mult(mTA, mTNS, mA);
        gmm::mult(mTM, mTNS, mM);
        gmm::mult(mOA, vTU, vOTU);
        gmm::mult(mOM, vFU, gmm::scaled(vOTU, -1),
           vFUt);
        gmm::add(vS, vFUt);
        gmm::mult(gmm::transposed(mNS), vFUt, vT);

        P.build_with(mA, 50, 1E-9);
        iter.init();
        iter.set_rhsnorm(residual);
        iter.set_noisy(1);
        iter.set_maxiter(40000);
        gmm::gmres(mA, vU, vT, P, 50, iter);
        converge = iter.converged();

        gmm::clean(vU, 1.0E-12);
        gmm::mult(mNS, vU, vTU, vFU);
```

```
            gmm::clean(vFU, 1.0E-12);
            exp.write_point_data(*mfu, vFU);
            exp.serie_add_object("option_price");
            expsl.write_point_data(*mfu, vFU);
            expsl.serie_add_object("option_price");


        }

        //start second period
        mymesh_o = mymesh;
        mymesh = new getfem::mesh;
        BuildMesh(false);
        mfu_o = mfu;
        mf_coe_o = mf_coe;
        mf_coe = new getfem::mesh_fem;
        mfu = new getfem::mesh_fem;
        mim_o = mim;
        mim = new getfem::mesh_im;
        BuildFEM(false);

        FindBoundary(cd, cu);
        AssembleMatrix();
        //get u at time t+1 from time t


        nb_dof = mfu->nb_dof();
        //derive the linear system for time 0
        gmm::scale(mA, -1);
        gmm::add(mS, mA);
        gmm::add(gmm::scaled(mM,rd), mA);
        gmm::scale(mM, 1/t_step);
        gmm::add(mM, mA);


        GetInitU(false);
        delete mymesh_o;
        delete mfu_o;
        delete mf_coe_o;
        delete mim_o;

        nb_col = gmm::vect_size(vU);
        //export the result as dx
        //
```

```
getfem::dx_export expsl_n("solution_sl_n.dx", true);
getfem::stored_mesh_slice sl_n;
sl_n.build(*mymesh,
          getfem::slicer_half_space(base_node(5,0.2),
                 base_node(0, 1), 0), 4);

expsl_n.exporting(sl_n);
expsl_n.write_point_data(*mfu, vFU);
expsl_n.serie_add_object("option_price");

getfem::dx_export expn("solution_second.dx", true);
expn.exporting(*mfu);

expn.write_point_data(*mfu, vFU);
expn.serie_add_object("option_price");

nb_col = gmm::vect_size(vU);
gmm::resize(mTA, nb_col, nb_dof);
gmm::resize(mOA, nb_dof, nb_dof);
gmm::resize(mOM, nb_dof, nb_dof);
gmm::resize(mTM, nb_col, nb_dof);
gmm::copy(mA, mOA);
gmm::copy(mM, mOM);
gmm::resize(mA, nb_col, nb_col);
gmm::resize(mM, nb_col, nb_col);


gmm::resize (mTNS, nb_dof, nb_col);
gmm::resize(vFUt, nb_dof);
gmm::resize(vFU, nb_dof);
gmm::clear(vFU);
gmm::clear(vFUt);

gmm::resize(vOTU ,nb_dof);
gmm::clear(vOTU);

gmm::resize (vT, nb_col);
gmm::clear(vT);

gmm::clean(vU, 1.0E-12);
gmm::mult(mNS, vU, vTU, vFU);
gmm::clean(vFU, 1.0E-12);
expn.write_point_data(*mfu, vFU);
expn.serie_add_object("option_price");
```

```
gmm::ilut_precond<sparse_matrix_type> PN;

for(i=1; i<= (t-0.2)/t_step; i++){
        t_time = t_step*i;
        nb_col = ApplyBoundary(t_time, cu);
        gmm::resize(mNS, nb_dof ,nb_col);

        gmm::mult(gmm::transposed(mNS), mOA, mTA);
        gmm::mult(gmm::transposed(mNS), mOM, mTM);
        gmm::copy(mNS, mTNS);
        gmm::mult(mTA, mTNS, mA);
        gmm::mult(mTM, mTNS, mM);
        gmm::mult(mOA, vTU, vOTU);
        gmm::mult(mOM, vFU, gmm::scaled(vOTU, -1),
            vFUt);
        gmm::add(vS, vFUt);
        gmm::mult(gmm::transposed(mNS), vFUt, vT);
        PN.build_with(mA, 50, 1E-9);
        iter.init();
        iter.set_rhsnorm(residual);
        iter.set_noisy(1);
        iter.set_maxiter(40000);
        gmm::gmres(mA, vU, vT, PN, 50, iter);
        converge = iter.converged();

        gmm::clean(vU, 1.0E-12);
        gmm::mult(mNS, vU, vTU, vFU);
        gmm::clean(vFU, 1.0E-12);
        expn.write_point_data(*mfu, vFU);
        expn.serie_add_object("option_price");
        expsl_n.write_point_data(*mfu, vFU);
        expsl_n.serie_add_object("option_price");
    }

    return converge;
}
```

B.3 Multi-dimensional Heston model

```
#include <iostream>
#include <string>
#include <math.h>
```

```
#include "heston_3d_fem.h"
#include "param.h"
#include <getfem/bgeot_poly_composite.h>
#include <getfem/bgeot_comma_init.h>
#include <getfem/getfem_regular_meshes.h>
#include <getfem/getfem_derivatives.h>
#include <getfem/getfem_superlu.h>
#include <getfem/getfem_export.h>
#include <getfem/getfem_import.h>
#include <getfem/getfem_mesh_slice.h>


using namespace std;

double coe_mu1_3d(const base_node &x){
        return (REUR-RJPY+A1*A1*sqrt(x[1])+A2*A2*sqrt(x[2])
           -0.5*A1*XI1*RHO1 - 0.5*A2*XI2*RHO2);
}


double coe_mu2_3d(const base_node &x){
        double kappa = 0, theta = 0;
        kappa = KAPPA1+ RHO1*XI1*1.6177;
        theta = THETA1*KAPPA1/kappa;
        return (kappa*(theta - x[1]) - 0.5*XI1*XI1);
}

double coe_mu3_3d(const base_node &x){
        double kappa = 0, theta = 0;
        kappa = KAPPA2+ RHO2*XI2*1.3588;
        theta = THETA2*KAPPA2/kappa;
        return (kappa*(theta - x[2]) - 0.5*XI2*XI2);
}


double coe_sigma11_3d(const base_node &x){

        return (0.5*x[1]*A1*A1 + 0.5*x[2]*A2*A2);
}

double coe_sigma12_3d(const base_node &x){
        return (0.5*x[1]*XI1*RHO1*A1);
}

double coe_sigma13_3d(const base_node &x){
```

```
        return (0.5*x[2]*XI2*RHO2*A2);
}

double coe_sigma22_3d(const base_node &x){
        return (0.5*x[1]*XI1*XI1);
}

double coe_sigma33_3d(const base_node &x){
        return (0.5*x[2]*XI2*XI2);
}

double init_con_st_3d(const base_node &x){
        if(exp(-x[0])> STRIKE_EUR && -x[0] < UP_B_EUR)
        {
                return (1 - STRIKE_EUR*exp(x[0]));
                }
        else return 0;
}

double neumann_boundary_con_3d(const base_node &x){
        return ((0.5*x[1]*A1*A1 + 0.5*x[2]*A2*A2)*exp(x[0]));
}

H3PDESolve::H3PDESolve(){
        dim = 3;
        cd = DOWN_B_EUR;
        cu = UP_B_EUR;
        k = -log(100);
        t = 0.5;
        rd = REUR;
        rf = RJPY;
        s_step = 0.1;
        t_step = 0.005;
        residual = 1E-15;
        it_number = 0;
        mymesh = new getfem::mesh;
        mfu = new getfem::mesh_fem;
        mf_coe = new getfem::mesh_fem;
        mim = new getfem::mesh_im;
        mesh_file = "gmsh:stv_3d.msh";
        mesh_file_n = "gmsh:stv_3d_n.msh";
}

H3PDESolve::~H3PDESolve(){
```

```
        delete mymesh;
        delete mfu;
        delete mf_coe;
        delete mim;
}

void H3PDESolve::BuildMesh(bool init){
        if(init)
                getfem::import_mesh(mesh_file, *mymesh);
        else getfem::import_mesh(mesh_file_n, *mymesh);
}

void H3PDESolve::FindBoundary(double down, double up){

        getfem::mesh_region border_faces;
        getfem::outer_faces_of_mesh(*mymesh, border_faces);

        for (getfem::mr_visitor i(border_faces);
                        !i.finished(); ++i) {
                assert(i.is_face());
                base_node un = mymesh->
                   normal_of_face_of_convex(i.cv(), i.f());
                un /= gmm::vect_norm2(un);
                if(un[0] < 0)//dirichlet boundary condition
                   when $e^{x} = c_{d}$
                {
                        mymesh->region(D_DIRICHLET_BOUNDARY_X)
                           .add(i.cv(), i.f());
                }
                else if(un[0] > 0)//neumann boundary condition
                    when $e^{x} = \infty$
                {
                        if(up == 0){ //means infinity
                                mymesh->region(
                                   D_NEUMANN_BOUNDARY_X).add
                                        (i.cv(), i.f());
                        }else
                        {
                                mymesh->region(
                                   D_DIRICHLET_BOUNDARY_X).add
                                   (i.cv(), i.f());

                        }
```

```
                }else if(gmm::abs(un[1]))
                {
                        mymesh->region(D_NEUMANN_BOUNDARY_V).
                            add(i.cv(), i.f());
                }
                else if(gmm::abs(un[2]))//neumann boundary
                    condition when $v=0, \infty$
                {
                        mymesh->region(D_NEUMANN_BOUNDARY_V).
                            add(i.cv(), i.f());
                }
        }

}

int H3PDESolve::ApplyBoundary(double time, double up){


        int nb_dof = 0, coe_nb_dof = 0;
        int i = 0;
        int nb_col = 0;
        nb_dof = mfu->nb_dof();
        coe_nb_dof = mf_coe->nb_dof();
        gmm::resize(vTU, nb_dof);
        gmm::clear(vTU);
        gmm::resize(mNS, nb_dof, nb_dof);
        gmm::clear(mNS);
        gmm::resize(vS, nb_dof);
        gmm::clear(vS);

        plain_vector vT(coe_nb_dof);
        gmm::clear(vT);
        plain_vector vR(nb_dof);
        col_sparse_matrix_type mH(nb_dof, nb_dof);
        plain_vector vK(dim*coe_nb_dof);
        gmm::clear(vK);

        if(up == 0.0)//up boundary = 0 means it is infinite
        {
                //neumman boundary condition
                getfem::interpolation_function(*mf_coe, vT,
                    neumann_boundary_con_3d);
                for(int j=0; j< coe_nb_dof; j++)
                {
```

```
                vK[dim*j] = vT[j]*exp(-1*rf*time) ;
            }
            asm_normal_source_term(vS, *mim, *mfu,
                                   *mf_coe, vK,
                                      D_NEUMANN_BOUNDARY_X);
    }
    gmm::clear(vT);
    asm_source_term(vS, *mim, *mfu, *mf_coe, vT,
       D_NEUMANN_BOUNDARY_V);
    gmm::clear(vT);

    getfem::asm_dirichlet_constraints(mH, vR, *mim, *mfu,
       *mf_coe, *mf_coe, vT, D_DIRICHLET_BOUNDARY_X);
            gmm::clean(mH, 1.0E-12);
            nb_col = getfem::Dirichlet_nullspace(mH, mNS,
                                                 vR, vTU);
            return nb_col;

    }

    void H3PDESolve::BuildFEM(bool init){
            int qdim = 0;
            mfu->init_with_mesh(*mymesh);
            mim->init_with_mesh(*mymesh);
            mf_coe->init_with_mesh(*mymesh);
            mfu->set_finite_element(
                            getfem::fem_descriptor("FEM_PK(3, 1)")
                               );
            mf_coe->set_finite_element(
                            getfem::fem_descriptor("FEM_PK(3, 1)")
                               );
            mim->set_integration_method(
                    getfem::int_method_descriptor("IM_TETRAHEDRON
                       (5)"));
            qdim = mfu->get_qdim();

    }

    void H3PDESolve::AsmGradBaseMatrix(sparse_matrix_type &M,
                    const getfem::mesh_im &mim,
                    const getfem::mesh_fem &mf,
                    const getfem::mesh_fem &mfdata,
                    const plain_vector &V,
                    const getfem::mesh_region
```

```
                &rg = getfem::mesh_region::all_convexes()){

        getfem::generic_assembly assem;
        assem.push_mi(mim);
        assem.push_mf(mf);
        assem.push_mf(mfdata);
        assem.push_data(V);
        assem.push_mat(M);
        if(mf.get_qdim() == 1)
                assem.set("a=data$1(mdim(#1), #2);"
                        "M$1(#1,#1)+=comp(Base(#1).Grad(#1).
                           Base(#2))(:,:,j,i).a(j, i)");
        assem.assembly(rg);

}

void H3PDESolve::AssembleMatrix(){
        int nb_dof = 0, coe_nb_dof = 0;
        int i = 0;

        coe_nb_dof = mf_coe->nb_dof();
        nb_dof = mfu->nb_dof();
        plain_vector vA(coe_nb_dof), vB(coe_nb_dof),
                        vC(coe_nb_dof), vT(coe_nb_dof);
        plain_vector vSI11(coe_nb_dof), vSI12(coe_nb_dof),
                        vSI22(coe_nb_dof), vSI13(coe_nb_dof),
                        vSI33(coe_nb_dof);
        plain_vector vMU(dim*coe_nb_dof);
        plain_vector vSI(dim*dim*coe_nb_dof);

        getfem::interpolation_function(*mf_coe,
                                        vA, coe_mu1_3d);
        getfem::interpolation_function(*mf_coe,
                                        vB, coe_mu2_3d);
        getfem::interpolation_function(*mf_coe,
                                        vC, coe_mu3_3d);
        for(i =0; i< coe_nb_dof; i++){
                vMU[dim*i] = vA[i];
                vMU[dim*i+1] = vB[i];
                vMU[dim*i+2] = vC[i];
        }
        gmm::resize(mA, nb_dof, nb_dof);
        gmm::clear(mA);
```

```
    AsmGradBaseMatrix(mA, *mim, *mfu, *mf_coe, vMU);
    getfem::interpolation_function(*mf_coe, vSI11,
        coe_sigma11_3d);
    getfem::interpolation_function(*mf_coe, vSI12,
        coe_sigma12_3d);
    getfem::interpolation_function(*mf_coe, vSI13,
        coe_sigma13_3d);
    getfem::interpolation_function(*mf_coe, vSI22,
        coe_sigma22_3d);
    getfem::interpolation_function(*mf_coe, vSI33,
        coe_sigma33_3d);
    for(i = 0; i<coe_nb_dof; i++){
            vSI[dim*dim*i] = vSI11[i];
            vSI[dim*dim*i+1] = vSI12[i];
            vSI[dim*dim*i+2] = vSI13[i];
            vSI[dim*dim*i+3] = vSI12[i];
            vSI[dim*dim*i+4] = vSI22[i];
            vSI[dim*dim*i+6] = vSI13[i];
            vSI[dim*dim*i+8] = vSI33[i];
        }
        gmm::resize(mS, nb_dof, nb_dof);
        gmm::clear(mS);
        asm_stiffness_matrix_for_scalar_elliptic(mS, *mim,
                                *mfu, *mf_coe, vSI);
        gmm::resize(mM, nb_dof, nb_dof);
        gmm::clear(mM);
        asm_mass_matrix(mM, *mim, *mfu, *mf_coe);

}

void H3PDESolve::GetInitU(bool init){
        int nb_dof = 0, coe_nb_dof = 0;
        int nb_col = 0;
        nb_dof = mfu->nb_dof();
        coe_nb_dof = mf_coe->nb_dof();
        plain_vector vT(coe_nb_dof);
        sparse_matrix_type mH(nb_dof, nb_dof);
        plain_vector vR(nb_dof);
        plain_vector vTR(nb_dof);

        sparse_matrix_type mtH(nb_dof, nb_dof);
        plain_vector vtR(nb_dof);

        if(init)
```

```
        {
                getfem::interpolation_function(*mf_coe, vT,
                    init_con_st_3d);
        }
        else
        {
                getfem::interpolation(*mfu_o, *mfu, vFU, vT,
                    0);
        }
        //derive the linear system HU = R
        asm_mass_matrix(mH, *mim, *mfu);
        asm_source_term(vR, *mim, *mfu, *mf_coe, vT);
        gmm::scale(mH, 1.0E+5);
        gmm::scale(vR, 1.0E+5);
        gmm::copy(mH, mtH);
        gmm::copy(vR, vtR);
        //apply the boundary condition
        nb_col = ApplyBoundary(0, cu);
        gmm::resize(mNS, nb_dof ,nb_col);
        gmm::mult(mH, vTU, gmm::scaled(vR, -1.0), vTR);
        gmm::resize(vU, nb_col);
        gmm::clear(vU);
        gmm::resize(vR, nb_col);
        gmm::clear(vR);

        gmm::mult(gmm::transposed(mNS),
                        gmm::scaled(vTR, -1.0), vR);
        sparse_matrix_type mTH(nb_col, nb_dof);
        gmm::mult(gmm::transposed(mNS), mH, mTH);
        gmm::resize(mH, nb_col, nb_col);
        sparse_matrix_type mTNS(nb_dof, nb_col);
        gmm::copy(mNS, mTNS);
        gmm::mult(mTH, mTNS, mH);
        //solving the system to get U at time 0
        gmm::iteration iter(residual, 1, 40000);
        gmm::ilut_precond<sparse_matrix_type> P(mH, 50, 1E-9);
        gmm::gmres(mH, vU, vR, P, 50,  iter);
        gmm::resize(vFU, nb_dof);
        gmm::clear(vFU);
        gmm::mult(mNS, vU, vTU, vFU);
        gmm::clean(vFU, 1.0E-12);

        gmm::copy(vT, vFU);
```

```
}

double H3PDESolve::solve(){
        double o_price = 0.0;
        int nb_dof = 0, nb_col = 0;
        int i = 0;
        double t_time = 0.0;
        int period = 2;

        char s[100];
        gmm::iteration iter(residual, 1, 40000);
        double time = dal::uclock_sec();

        double converge = 0;
        BuildMesh(true);
        FindBoundary(cd, cu);
        BuildFEM(true);
        mymesh->write_to_file("stv_mesh_3d.msh");

        AssembleMatrix();

        nb_dof = mfu->nb_dof();
        //derive the linear system for time 0
        gmm::scale(mA, -1);
        gmm::add(mS, mA);
        gmm::add(gmm::scaled(mM,rd), mA);
        gmm::scale(mM, 1/t_step);
        gmm::add(mM, mA);

        //apply the boundary and get additional source term
        GetInitU(true);

        nb_col = gmm::vect_size(vU);
        //export the result as dx
        getfem::dx_export expsl("solution_sl.dx", true);
        getfem::stored_mesh_slice sl;
        sl.build(*mymesh, getfem::slicer_half_space(base_node
           (-5, 0.2, 0.3), base_node(0, 1, 0), 0),
         getfem::slicer_half_space(base_node(-5,0.2, 0.3),
                 base_node(0, 0, 1), 0), 4);
        expsl.exporting(sl);
        expsl.write_point_data(*mfu, vFU);
        expsl.serie_add_object("option_price");
```

```
        getfem::dx_export exp("solution.dx", true);
        exp.exporting(*mfu);

        exp.write_point_data(*mfu, vFU);
        exp.serie_add_object("option_price");

        plain_vector vFUt(nb_dof);
        gmm::clear(vFUt);

        plain_vector vT(nb_col);
        gmm::clear(vT);

        plain_vector vOTU(nb_dof);
        gmm::clear(vOTU);

        sparse_matrix_type mTA(nb_col, nb_dof);
        sparse_matrix_type mOA(nb_dof, nb_dof);
        sparse_matrix_type mOM(nb_dof, nb_dof);
        sparse_matrix_type mTM(nb_col, nb_dof);
        sparse_matrix_type mTNS(nb_dof, nb_col);
        gmm::copy(mA, mOA);
        gmm::copy(mM, mOM);
        gmm::resize(mA, nb_col, nb_col);
        gmm::resize(mM, nb_col, nb_col);
        gmm::mult(gmm::transposed(mNS), mOA, mTA);
        gmm::mult(gmm::transposed(mNS), mOM, mTM);
        gmm::copy(mNS, mTNS);
        gmm::mult(mTA, mTNS, mA);
        gmm::mult(mTM, mTNS, mM);

        gmm::ilut_precond<sparse_matrix_type> P;
        P.build_with(mA, 50, 1E-9);
        for(i=1; i<= 0.2/t_step; i++){
                t_time = t_step*i;
                gmm::mult(mOA, vTU, vOTU);
                gmm::mult(mOM, vFU, gmm::scaled(vOTU, -1),
                   vFUt);
                gmm::add(vS, vFUt);
                gmm::mult(gmm::transposed(mNS), vFUt, vT);

                iter.init();
                iter.set_rhsnorm(residual);
                iter.set_noisy(1);
                iter.set_maxiter(40000);
```

```
            gmm::gmres(mA, vU, vT, P, 50, iter);
            converge = iter.converged();

            gmm::clean(vU, 1.0E-12);
            gmm::mult(mNS, vU, vTU, vFU);
            gmm::clean(vFU, 1.0E-12);
            exp.write_point_data(*mfu, vFU);
            exp.serie_add_object("option_price");
            expsl.write_point_data(*mfu, vFU);
            expsl.serie_add_object("option_price");

    }
    //start second period
    mymesh_o = mymesh;
    mymesh = new getfem::mesh;
    BuildMesh(false);
    mfu_o = mfu;
    mf_coe_o = mf_coe;
    mf_coe = new getfem::mesh_fem;
    mfu = new getfem::mesh_fem;
    mim_o = mim;
    mim = new getfem::mesh_im;
    BuildFEM(false);

    FindBoundary(cd, cu);
    AssembleMatrix();

    nb_dof = mfu->nb_dof();
    //derive the linear system for time 0
    gmm::scale(mA, -1);
    gmm::add(mS, mA);
    gmm::add(gmm::scaled(mM,rd), mA);
    gmm::scale(mM, 1/t_step);
    gmm::add(mM, mA);

    //get u at time t+1 from time t
    GetInitU(false);
    delete mymesh_o;
    delete mfu_o;
    delete mf_coe_o;
    delete mim_o;

    nb_col = gmm::vect_size(vU);
    //export the result as dx
```

```
getfem::dx_export expsl_n("solution_sl_n.dx", true);
getfem::stored_mesh_slice sl_n;
sl_n.build(*mymesh, getfem::slicer_half_space(
   base_node(-5, 0.2, 0.3), base_node(0, 1, 0), 0),
 getfem::slicer_half_space(base_node(-5,0.2, 0.3),
        base_node(0, 0, 1), 0), 4);


expsl_n.exporting(sl_n);
expsl_n.write_point_data(*mfu, vFU);
expsl_n.serie_add_object("option_price");

getfem::dx_export expn("solution_second.dx", true);
expn.exporting(*mfu);

expn.write_point_data(*mfu, vFU);
expn.serie_add_object("option_price");

gmm::resize(mTA, nb_col, nb_dof);
gmm::resize(mOA, nb_dof, nb_dof);
gmm::resize(mOM, nb_dof, nb_dof);
gmm::resize(mTM, nb_col, nb_dof);
gmm::copy(mA, mOA);
gmm::copy(mM, mOM);
gmm::resize(mA, nb_col, nb_col);
gmm::resize(mM, nb_col, nb_col);


gmm::resize (mTNS, nb_dof, nb_col);
gmm::resize(vFUt, nb_dof);
gmm::clear(vFUt);

gmm::resize(vOTU ,nb_dof);
gmm::clear(vOTU);

gmm::resize (vT, nb_col);
gmm::clear(vT);

gmm::mult(gmm::transposed(mNS), mOA, mTA);
gmm::mult(gmm::transposed(mNS), mOM, mTM);
gmm::copy(mNS, mTNS);
gmm::mult(mTA, mTNS, mA);
gmm::mult(mTM, mTNS, mM);
```

```
gmm::ilut_precond<sparse_matrix_type> PN;
PN.build_with(mA, 50, 1E-9);

for(i=1; i<= (t-0.2)/t_step; i++){
        t_time = t_step*i;

        gmm::mult(mOA, vTU, vOTU);
        gmm::mult(mOM, vFU, gmm::scaled(vOTU, -1),
           vFUt);
        gmm::add(vS, vFUt);
        gmm::mult(gmm::transposed(mNS), vFUt, vT);

        iter.init();
        iter.set_rhsnorm(residual);
        iter.set_noisy(1);
        iter.set_maxiter(40000);
        gmm::gmres(mA, vU, vT, PN, 50, iter);
        converge = iter.converged();

        gmm::clean(vU, 1.0E-12);
        gmm::mult(mNS, vU, vTU, vFU);
        gmm::clean(vFU, 1.0E-12);
        expn.write_point_data(*mfu, vFU);
        expn.serie_add_object("option_price");
        expsl_n.write_point_data(*mfu, vFU);
        expsl_n.serie_add_object("option_price");
    }
    return converge;
}
```

Bibliography

[1] M. Ainsworth and J. T. Oden. *A posteriori error estimation in finite element analysis.* Wiley-Interscience, 2011.

[2] H. Albrecher, P. Mayer, W. Schoutens, and J. Tistaert. The little Heston trap. *Wilmott*, pages 83–92, 2006.

[3] G. Armstrong. Valuation formulae for window barrier options. *Applied Mathematical Finance*, 8(4):197–208, 2001.

[4] M. Behr. Simplex space–time meshes in finite element simulations. *International journal for numerical methods in fluids*, 57(9):1421–1434, 2008.

[5] F. Black and M. Scholes. The pricing of options and corporate liabilities. *The journal of political economy*, pages 637–654, 1973.

[6] P. Boyle and S. Lau. Bumping up against the barrier with the binomial method. *The Journal of Derivatives*, 1(4):6–14, 1994.

[7] P. Boyle and Y. Tian. An explicit finite difference approach to the pricing of barrier options. *Applied Mathematical Finance*, 5(1):17–43, 1998.

[8] P. Carr. Two extensions to barrier option valuation. *Applied Mathematical Finance*, 2(3):173–209, 1995.

[9] T. Cheuk and T. Vorst. Complex barrier options. *The Journal of Derivatives*, 4(1):8–22, 1996.

[10] I. J. Clark. *Foreign Exchange Option Pricing: A practitioner's Guide.* John Wiley & Sons, Chichester, 2011.

[11] A. Coddington and N. Levinson. *Theory of ordinary differential equations.* International series in pure and applied mathematics. McGraw-Hill, 1955.

[12] A. De Col, A. Gnoatto, and M. Grasselli. Smiles all around: FX joint calibration in a multi-heston model. *arXiv:1201.1782v2*, 2012.

[13] B. Dupire. Pricing with a smile. *Risk*, 7(1):18–20, 1994.

[14] L. Evans. *Partial Differential Equations*. Graduate Studies in Mathematics Series. Amer Mathematical Society, 2010.

[15] P. M. N. Feehan and C. A. Pop. Schauder a priori estimates and regularity of solutions to degenerate-elliptic linear second-order partial differential equations. *arXiv:1210.6727v1*, 2012.

[16] P. M. N. Feehan and C. A. Pop. Stochastic representation of solutions to degenerate elliptic and parabolic boundary value and obstacle problems with Dirichlet coundary conditions. *arXiv:1204.1317v4*, 2013.

[17] S. Figlewski and B. Gao. The adaptive mesh model: a new approach to efficient option pricing. *Journal of financial economics*, 53(3):313–351, 1999.

[18] P. A. Forsyth, K. R. Vetzal, and R. Zvan. A finite element approach to the pricing of discrete lookbacks with stochastic volatility. *Applied Mathematical Finance*, 6(2):87–106, 1999.

[19] H. Geman and M. Yor. Pricing and hedging double-barrier options: A probabilistic approach. *Mathematical finance*, 6(4):365–378, 1996.

[20] C. Geuzaine and J.-F. Remacle. Gmsh: A three-dimensinal finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.

[21] P. Glasserman and J. Staum. Conditioning on one-step survival for barrier option simulations. *Operations Research*, 49(6):923–937, 2001.

[22] D. Heath and M. Schweizer. Martingales versus pdes in finance: an equivalence result with examples. *Journal of Applied Probability*, 37(4):947–957, 2000.

[23] S. Heston. A closed-form solution for options with stochastic volatility with applications to bond and currency options. *Review of financial studies*, 6(2):327–343, 1993.

[24] R. C. Heynen and H. M. Kat. Discrete partial barrier options with a moving barrier. *The Journal of Financial Engineering*, 5(3):199–209, 1996.

[25] Y. Hilpisch. Calibrating Heston's stochastic volatility model. http://www.dexision.org/dok/Visixion_Calibrating_Heston.pdf, 2009.

[26] T. Hughes and G. Hulbert. Space-time finite element methods for elastodynamics: formulations and error estimates. *Computer methods in applied mechanics and engineering*, 66(3):339–363, 1988.

[27] J. Hull. *Options, Futures, & Other Derivatives.* The Prentice Hall Series in Finance. Prentice Hall, 2009.

[28] J. Jia, T. Lu, K. Wang, H. Wang, and Y. Ren. A uniformly optimal-order error estimate for a bilinear finite element method for degenerate convection-diffusion equations. *Numerical Methods for Partial Differential Equations*, 28(3):768–781, 2012.

[29] I. Karatzas and S. E. Shreve. *Brownian motion and stochastic calculus (second edition)*, volume 113. Springer Verlag, 1998.

[30] R. Kimmel and Y. Aït-Sahalia. Maximum likelihood estimation of stochastic volatility models. *Journal of Financial Economics*, 83(2):413–452, 2007.

[31] F. C. Klebaner. *Introduction to Stochastic Calculus with Applications (2nd Edition).* Imperial College Press, 2005.

[32] P. Knabner and L. Angerman. *Numerical Methods for Elliptic and Parabolic Partial Differential Equations.* Texts in Applied Mathematics. Springer, 2010.

[33] N. Kunitomo and M. Ikeda. Pricing options with curved boundaries. *Mathematical finance*, 2(4):275–298, 1992.

[34] V. Mazia. *Sobolev Spaces: With Applications to Elliptic Partial Differential Equations.* Grundlehren Der Mathematischen Wissenschaften. Springer Verlag, 2011.

[35] R. Merton. Theory of rational option pricing. *The Bell Journal of Economics and Management Science*, pages 141–183, 1973.

[36] S. Metwally and A. Atiya. Using brownian bridge for fast simulation of jump-diffusion processes and barrier options. *The journal of derivatives*, 10(1):43–54, 2002.

[37] A. Mijatović and M. Urusov. On the martingale property of certain local martingales. *arXiv:0905.3701v3*, 2010.

[38] S. Mikhailov and U. Nogel. Hestons stochastic volatility model: Implementation, calibration and some extensions. *The Best of Wilmott*, page 401, 2003.

[39] E. V. Radkevich. Equations with nonnegative characteristic form. *Journal of Mathematical Sciences*, 158(3):297–452, 2009.

[40] E. V. Radkevich. Equations with nonnegative characteristic form II. *Journal of Mathematical Sciences*, 158(4):453–604, 2009.

[41] D. Reiswich and U. Wystup. A guide to FX options quoting conventions. *The Journal of Derivatives*, 18(2):58–68, 2010.

[42] Y. Renard and J. Pommier. Getfem++ c++ library. http://download.gna.org/getfem/html/homepage, 2010.

[43] D. Rich. The mathematical foundations of barrier option-pricing theory. *Advances in Futures and Options Research*, 7, 1994.

[44] P. Ritchken. On pricing barrier options. *Currency Derivatives: Pricing Theory, Exotic Options, and Hedging Applications*, 12:275, 1998.

[45] L. G. Rogers and O. Zane. Valuing moving barrier options. *Journal of Computational Finance*, 1(1):5–11, 1997.

[46] M. Rubinstein and E. Reiner. Breaking down the barriers. *Risk*, 4:28–35, 1991.

[47] X. Tao and S. Zhang. Boundary unique continuation theorems under zero Neumann boundary conditions. *Bull. Austral. Math. Soc.*, 72(1):67–85, 2005.

[48] V. Thomée. *Galerkin Finite Element Methods for Parabolic Problems*. Springer Series in Computational Mathematics. Springer London, Limited, 2006.

[49] B. Tomas. *Arbitrage Theory in Continuous Time.* Oxford University Press, New York, 2009.

[50] J. Topper. Finite element modelling of exotic options. *Available at SSRN 150814*, 1998.

[51] P. Wilmott, J. DeWynne, and S. Howison. *Option Pricing: Mathematical Models and Computation.* Oxford Financial Press, 1993.

[52] G. Winkler, T. Apel, and U. Wystup. Valuation of options in Heston's stochastic volatility model using finite element methods. *Foreign Exchange Risk. Risk Publications, London*, pages 278–313, 2001.

[53] Z. Wu, J. Yin, and C. Wang. *Elliptic & Parabolic Equations.* World Scientific, 2006.

[54] E. Zachmanoglou and D. Thoe. *Introduction to Partial Differential Equations With Applications.* Dover Books on Mathematics Series. Dover Publications, Incorporated, 1986.

[55] Z. Zhu and N. Stokes. A finite element platform for pricing path dependent exotic options. *Proc. QMF99, Sydney*, 1999.

[56] R. Zvan, P. A. Forsyth, and K. R. Vetzal. A general finite element approach for PDE option pricing models. *Methods*, 19:3, 1998.

[57] R. Zvan, K. R. Vetzal, and P. A. Forsyth. PDE methods for pricing barrier options. *Journal of Economic Dynamics and Control*, 24(11):1563–1590, 2000.